CodeWarrior®
PowerPlant Advanced Topics

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Metrowerks CodeWarrior copyright ©1993–1997 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 2201 Donley Drive, Suite 310 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Mail order	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction 11
What's in PowerPlant Advanced Topics 11
What's New in PowerPlant Advanced Topics 11
What You Should Know 12
Chapter Organization 12
Starting Points 13

2 Drag and Drop in PowerPlant 15
Introduction to Drag and Drop in PowerPlant 15
Drag and Drop Strategy 16
Drag and Drop Classes 18
LDragTask 19
LDropArea 20
LDragAndDrop 23
Implementing Drag and Drop in PowerPlant 24
Looking for the Drag Manager 24
Handling Clicks 25
Identifying a Drag 28
Creating a Drag Task 28
The simple approach 29
The flexible approach 29
Tracking a Drag 31
Receiving a Drop 32
Providing Custom Drag Behavior 32
Summary of Drag and Drop in PowerPlant 34
Code Exercise for Drag and Drop 35

3 Threads in PowerPlant 45
Introduction to Threads in PowerPlant 45
The Thread Strategy 46
Thread States 47
Semaphores 49
Inter-Thread Communication 50
Thread Classes 50

LThread 51
LThread attributes 52
LThread Behaviors 53
The Run() function 55
LSimpleThread 57
UMainThread 58
LYieldAttachment 58
LSemaphore 59
LEventSemaphore 60
LMutexSemaphore 60
StMutex 60
StCritical 61
LLink 61
LQueue 61
LSharedQueue 62
Implementing Threads in PowerPlant 63
Initializing Threads 63
Determine if the Thread Manager is present 63
Create the main thread 64
Creating Threads 65
Allocating memory for a thread 65
Calling the LThread constructor 65
Running a Thread 67
Modifying Thread State 68
Deleting Threads 69
Data Coherency 71
Criticality 71
Context Switching 72
Using Semaphores 74
Inter-Thread Communication 78
Asynchronous Operations 81
Summary of Threads in PowerPlant 87
Code Exercise for Threads 88

4 Actions in PowerPlant 97
Introduction to Actions in PowerPlant 97

The Undo Strategy 98
Action Classes 98
LCommander 99
LAction 100
LUndoer 102
LTETextAction 103
Implementing Undo in PowerPlant 104
Create Action Classes 104
Attach an Undoer 105
Post an Action 105
Implement Multilevel Undo 106
Summary of Undo in PowerPlant 107
Code Exercise for Actions 107

5 Tables in PowerPlant 115
Introduction to Tables in PowerPlant 115
Table Strategy 116
Table Architecture 116
General Table Implementation 118
Table Classes 119
STableCell 122
LTableView 123
LTableView Services 124
Row, column, and cell management 125
Accessing cells. 125
Cell geometry 126
Cell selection 126
Data storage. 127
Drawing and clicking. 128
LColumnView 129
LTextColumn 130
LSmallIconTable 130
LHierarchyTable 130
LTextHierTable. 133
LTableGeometry 133
LTableMonoGeometry 135

LTableMultiGeometry 135
LTableSelector 136
LTableSingleSelector 137
LTableMultiSelector 137
LTableStorage 138
LTableArrayStorage 139
LCollapsableTree 140
LNodeArrayTree 141
LDropFlag 141
Implementing Tables in PowerPlant 141
Creating a Table 142
Creating helper objects 142
Adding rows and columns 143
Managing Rows and Columns 143
Inserting columns 143
Inserting rows 144
Removing rows and columns 145
Changing row and column size 146
Setting Cell Data 146
Getting Cell Data 147
Handling Clicks in a Cell 148
Responding to Selections 149
Drawing a Cell 149
Drawing a cell 149
Nesting level 150
The Drop Flag 150
Highlighting a cell 150
Refreshing a cell 151
Finding Cells 151
Finding Data in a Table 152
Scrolling a Table 153
Summary of Tables in PowerPlant 153
Code Exercise for Tables 154

6 Apple Events in PowerPlant 167
Introduction to Apple Events in PowerPlant 167

Where to Learn More About Apple Events . . . 168
Apple Event Strategy . . . 169
Apple Event Classes . . . 171
LModelObject . . . 172
Managing elements . . . 174
Handling properties . . . 177
Handling events . . . 178
LModelDirector . . . 180
LModelProperty . . . 180
UExtractFromAEDesc . . . 180
StAEDescriptor . . . 182
UAEDesc . . . 184
UAppleEventsMgr . . . 185
Apple Event Resources . . . 186
The 'aete' Resource . . . 186
The 'aedt' Resource . . . 188
Editing Apple Event Resources . . . 188
Implementing Apple Events in PowerPlant . . . 188
Adding Classes . . . 189
Adding Properties . . . 191
Adding Custom Apple Events . . . 191
Beyond the Basics . . . 192
Laziness . . . 192
Default submodels . . . 193
UAEGizmos . . . 193
Whose clauses . . . 194
Recordability . . . 194
Code Exercise for Apple Events . . . 194
Edit Apple Event Resources . . . 196
Create a Model Object in the Application . . . 204
Add Model Properties to the Class . . . 207
Add a Custom Event to the Application . . . 210
Improve HandleCreateElementEvent() . . . 212

7 Networking in PowerPlant . . . 217
Introduction to Networking in PowerPlant . . . 217

Where to Learn More About Networking 219
Software Requirements 219
Networking Strategy 221
Generic Network Interface. 221
Other Classes 222
Strategic Summary 222
Networking Classes. 223
UNetworkFactory 223
LInternetAddress 225
LTCPEndpoint 227
LUDPEndpoint 231
Implementing a Network-Savvy Application 234
Creating a Client 235
Obtaining an Address. 236
Creating a Client Endpoint 236
Binding to a Local Port 236
Connecting to a Server 237
Sending Data 238
Receiving Data. 238
Disconnecting from a Server 238
Handling a Disconnect Request 239
Creating a Server. 239
Listening for Incoming Connections 240
Responding to Incoming Connections. 241
Implementing Threads 242
Connectionless Datagram Communications 243
Summary of Networking in PowerPlant 243
Code Exercise for Networking 244
SimpleClient. 244
SimpleServer 249

8 Internet Programming in PowerPlant. 253
Introduction to Internet Programming in PowerPlant 253
Where to Learn More About Internet Protocols 254
Software Requirements 255
Internet Programming Strategy 256

Generic Internet Protocol Interface 256
Specific Internet Protocol Interfaces 257
Internet Messages 258
General Utilities 260
Strategic Summary 260
Internet Classes 261
LInternetProtocol. 263
LSMTPConnection 266
LPOP3Connection 267
LHTTPConnection 271
LFTPConnection 272
LInternetResponse 276
LSMTPResponse 278
LPOP3Response 278
LHTTPResponse 279
LFTPResponse. 280
LInternetMessage 280
LMailMessage 281
LHTTPMessage 283
Internet Class Utilities 285
LDynamicBuffer 285
LHeaderField 286
Other Classes 287
Implementing an Internet Enabled Application 287
Choosing a Protocol 289
Creating a Protocol Client 290
Preparing Content 290
Addressing the Remote Computer 291
Creating the Protocol Thread. 291
Creating a Connection 293
Sending Content to a Server 294
Receiving Responses From a Server 294
Listening For Progress Messages 295
Closing Down a Connection 297
Handling Abnormal Conditions 297
Summary of Internet Protocol Usage in PowerPlant 298

Code Exercise 299

9 Offscreen Drawing in PowerPlant 313
Introduction to Offscreen Drawing in PowerPlant 313
Offscreen Drawing Strategy 314
Offscreen Drawing Classes 315
LGWorld 315
LGWorld Limitations 318
StOffscreenGWorld 319
LOffscreenView 321
Implementing Offscreen Drawing in PowerPlant 322
Using LOffscreenView 323
Using StOffscreenGWorld Directly 323
Using LGWorld 324
Code Exercise for Offscreen Drawing 325

10 Profiling in PowerPlant. 339
Introduction to Profiling in PowerPlant 339
Profiling Strategy 339
Rules of Thumb 340
Profiling Class 341
StProfileSection 342
~StProfileSection 343
Profiling Your Code 343
Code Exercise for Profiling 344
Profiling a Single Routine 345
Profiling an Application 349

Index 351

1

Introduction

This manual is a collection of topics designed to help you implement advanced features of PowerPlant and Mac OS in your code.

What's in PowerPlant Advanced Topics

Welcome to the *PowerPlant Advanced Topics* manual. This manual shows you how to use PowerPlant to implement a wide variety of advanced features of the Mac OS. There is a chapter for each topic. Each chapter begins with an introduction. They are:

- Introduction to Drag and Drop in PowerPlant
- Introduction to Threads in PowerPlant
- Introduction to Actions in PowerPlant
- Introduction to Tables in PowerPlant
- Introduction to Apple Events in PowerPlant
- Introduction to Networking in PowerPlant
- Introduction to Internet Programming in PowerPlant
- Introduction to Offscreen Drawing in PowerPlant
- Introduction to Profiling in PowerPlant
- Introduction to PowerTalk in PowerPlant

What's New in PowerPlant Advanced Topics

The chapters on internet programming iscompletely new. Other chapters have been revised slightly for minor updates and corrections.

There are additional topics in PowerPlant programming that we hope to address in future releases, including the PowerParts classes for OpenDoc support.

What You Should Know

To get the most from this manual, you should be familiar with PowerPlant programming. PowerPlant is a Mac OS application framework written in C++. Therefore you should be familiar with Mac OS programming, and with the C++ language.

In addition, this manual assumes you are familiar with the material covered in *The PowerPlant Book.* That title is available as part of the CodeWarrior documentation. *The PowerPlant Book* introduces you to PowerPlant and how to write a PowerPlant application. It covers basic PowerPlant features such as framework architecture, framework design, panes, views, controls, debugging, menus, windows, dialogs, file I/O, printing, periodicals, and attachments.

Chapter Organization

Most chapters in this manual follow the same internal structure. The first part of each chapter discusses PowerPlant fundamentals for the topic at hand. It introduces you to the classes involved in that chapter and how you use them. Each chapter discusses the relevant member functions and data members, the inheritance chain, and the common situations in which you use each class.

The second part of each chapter is a code exercise. In this part of the chapter you write real PowerPlant code following step-by-step instructions. This gives you an opportunity for hands-on practice with the real thing.

The code exercises are all application-based. PowerPlant is first and foremost an *application* framework, after all. However, you can use PowerPlant classes for other programming projects such as code resources and shared libraries.

Starting Points

Pick a topic that interests you, and go to that chapter. You do not need to read this manual sequentially. Each chapter is an independent entity. In each chapter you will learn what you need to know to use PowerPlant effectively to implement your chosen functionality.

2

Drag and Drop in PowerPlant

This chapter discusses how to implement drag and drop features in a PowerPlant application.

Introduction to Drag and Drop in PowerPlant

PowerPlant provides wrappers for the Mac OS Drag Manager, and implements most of the required functionality at a default level. If you use the PowerPlant default behavior, implementing drag and drop can be fairly simple. You also have the option of extending PowerPlant for your own purposes.

You'll see how as we discuss:

- Drag and Drop Strategy—the PowerPlant approach to drag and drop
- Drag and Drop Classes—the individual PowerPlant classes involved in drag and drop
- Implementing Drag and Drop in PowerPlant—working with PowerPlant's drag and drop classes

The code exercise at the end of the chapter takes you through the process of implementing drag and drop in real code.

This chapter does not teach you the intricacies of the Drag Manager, or the drag and drop human interface. For more information, consult the official Drag Manager documentation: "*Drag Manager Programmer's Guide*" and "*Drag and Drop H.I. Guidelines*." Both documents are available on the CodeWarrior Reference CD.

Drag and Drop Strategy

The Drag Manager supports drag operations at the window level. A window has a drag tracking handler to handle the visual feedback required as a drag passes across a window's content area. A window also has a drag receive handler to accept the data in the drag and add it to the window's content. Both of these handlers are callback routines used by the Mac OS to implement drag and drop functionality.

Because it works at the window level, the Mac OS Drag Manager has a somewhat coarse resolution. An individual window may have several different sections that display different kinds of data. For example, a window may have a text area, a drawing area, some buttons or controls, a tool area, and so forth. You may want to handle drag tracking and receiving in different ways in different portions of a window.

You can do this with the Drag Manager only if you test the location of the drag at any moment, compare that against the geography of your window contents, and then act accordingly. This kind of complex, location-dependent system is anathema to the principles of object-oriented programming.

PowerPlant extends the Drag Manager so that it becomes an object-oriented tool. PowerPlant implements the concept of a "drop area." In PowerPlant, you add drag and drop functionality to individual panes. The pane could be a window (windows are views, and views are panes), a scrolling view, a text view, a button, or any other pane-based visual element in PowerPlant. Any pane that is drag and drop aware is a drop area. Because a drag always involves a visual item, all drag operations in PowerPlant are pane-related.

There are two important architectural features that determine the internal workings of a drag-aware PowerPlant application. A PowerPlant application has a single tracking and receive handler installed with the Drag Manager. Nevertheless, each drop area has its own internal tracking and receiving handlers.

Here's how it works. PowerPlant maintains a list of all drop areas. The tracking and receive handlers installed for the Drag Manager

determine which drop area is involved in the operation, and dispatch control to the correct handler for the particular drop area. As a result, you are no longer limited to window-level resolution. You have object-level resolution for drag and drop functionality. You can define the behavior of each individual class of object with respect to drag and drop, not just each window.

In terms of implementation, a drag operation has a beginning, middle, and end. It begins when the user clicks and begins to drag an item. It continues while the user keeps the mouse button down. It ends when the user releases the mouse button.

The beginning occurs in a pane's `Click()` function when you write code to test for a drag. When you detect a drag, you create a drag task object. This may be an LDragTask object, or an object from a custom class derived from LDragTask. The drag task contains the data that represents the item or items being moved by the user. The drag task also contains a function for initiating the drag.

After creating the drag task object, you tell the drag task to begin tracking the drag. As the drag crosses various panes, you provide feedback to the user during the drag. The pane is responsible for providing the visual feedback during the drag.

A pane involved in a drag and drop operation should inherit from LDragAndDrop. LDragAndDrop is a mix-in class that implements the default behavior required to track a drag and receive a drop. LDragAndDrop is a concrete class that inherits from the abstract class LDropArea.

In a traditional implementation, you might have your window class inherit from LDragAndDrop. Then the user can drop items in your window. However, the PowerPlant strategy allows you to implement drag and drop for any individual pane as well.

When the user releases the mouse button at the end of a drag, the drop area receives the contents of the drop. Once again, it is LDragAndDrop that provides the functionality at the pane level. Any pane that inherits from LDragAndDrop may receive a drop.

PowerPlant provides functions that you override to put data into a drag, and to receive data from a drag. PowerPlant calls these functions at the appropriate moments. You provide the data and receive the data. PowerPlant does most of the rest of the background work.

Let's look at the features of the LDragTask, LDropArea, and LDragAndDrop classes to see how they work.

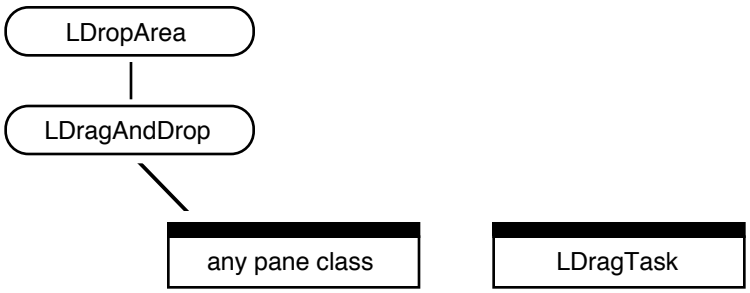
Drag and Drop Classes

There are three classes involved in PowerPlant's implementation of drag and drop functionality. They are:

- LDragTask—creates the drag record and starts a drag
- LDropArea—abstract class that provides the function interface for tracking and receiving a drag
- LDragAndDrop—a concrete implementation of LDropArea

Figure 2.1 illustrates the class hierarchy for these classes.

Figure 2.1 Drag and drop class hierarchy



If you wish to support drag and drop in your code, you create custom pane classes. Remember that views are also panes. In a typical implementation, it is a view class that inherits from LDragAndDrop. For example, you might create a text view class that inherits from both LTextEdit and LDragAndDrop.

LDragTask

When the user begins a drag, you create an LDragTask object. LDragTask class has three data members, as shown in Table 2.1.

Table 2.1 LDragTask data members

Data member	Stores
`mDragRef`	reference to this drag record
`mDragRegion`	region handle for the drag region
`mEventRecord`	reference to the event that started the drag

The `mDragRef` member stores a `DragReference` value. The Drag Manager creates and uses a `DragReference` to access information about the drag.

The `mDragRegion` member stores a handle to the drag region. When tracking a drag, the Drag Manager draws the region that defines the bounds of objects being dragged.

The `mEventRecord` member stores a pointer to an `EventRecord`. The event record contains the location where the drag started, in global coordinates.

LDragTask is a simple class. The most useful functions are shown in Table 2.2.

Table 2.2 LDragTask functions

Function	Purpose
`AddFlavors()`	add data to the drag in various flavors
`MakeDragRegion()`	create a drag region
`AddRectDragItem()`	add a rectangle to the drag region
`DoDrag()`	initiate the drag

The `AddFlavors()` and `MakeDragRegion()` functions give you the opportunity to add data in a variety of flavors to a drag, and to specify the precise drag region you wish to use. These functions are empty in LDragTask. We'll discuss when and how to implement them in "Creating a Drag Task" on page 28.

The `AddRectDragItem()` function is a utility routine that adds the bounds of the `Rect` you provide to an accumulating drag region.

The `DoDrag()` function is the routine you call to initiate the drag. The default implementation calls `AddFlavors()`, `MakeDragRegion()`, and the Drag Manager's `TrackDrag()` routine.

LDropArea

LDropArea is an abstract base class that provides the interface you use to implement drag tracking and receiving functionality.

LDropArea uses the LArray class. Other than that single exception, LDropArea—like many other independent modules in PowerPlant—does not requires any other PowerPlant classes. You can use LDropArea independently as a wrapper class for the Drag Manager. However, its most common use is as an integral part of a PowerPlant application through LDragAndDrop, its concrete descendant.

Most of the data members in LDropArea are used internally by PowerPlant. You won't have need to use them yourself. Table 2.3 lists the important data members.

Table 2.3 LDropArea data members

Data member	Stores
`mDragWindow`	`WindowPtr` for the window containing drop area
`mCanAcceptCurrentDrag`	whether the drop area can receive this drag
`mIsHilited`	whether the drop area is currently highlighted

Data member	Stores
`sDragHasLeftSender`	whether the drag has left the sender window
`sCurrentDropArea`	current drop area

Notice that the `sDragHasLeftSender` and `sCurrentDropArea` are static variables, so all instances of LDropArea share the same values.

LDropArea also defines a series of static class functions, some of which are shown inTable 2.4.

Table 2.4 LDropArea static functions

Function	Purpose
`DragAndDropIs-Present()`	returns `true` if the Mac OS Drag Manager is available
`AddDropArea()`	add a drop area to the list
`RemoveDropArea()`	remove a drop area from the list
`FindDropArea()`	determine drop area involved in the drag
`InstallHandlers()`	install tracking and receive handlers
`HandleDragTracking()`	drag tracking callback routine
`HandleDragReceive()`	drag receive callback routine
`InTrackingWindow()`	track a drag while it is in a window

These functions provide the core of PowerPlant's default implementation of drag and drop. These functions are used internally by PowerPlant. You should never have to call or modify these functions.

The `DragAndDropIsPresent()` function can be called to determine if the Drag Manager is present. PowerPlant checks at startup. This returns the result of that check.

The "drop area" functions manage PowerPlant's drop area list.

The two "handle" functions are Drag Manager callback routines. The `InstallHandlers()` function installs these tracking and receive handlers for the Drag Manager as the only handlers for the application. When tracking a drag inside a window, the default tracking handler calls `InTrackingWindow()`. The `InTrackingWindow()` code finds the correct drop area from the drop area list, and calls the correct handler.

When it is time to receive a drop, the default receive handler sends a `DoDragReceive()` message to the current drop area.

There are additional functions for custom drag behavior such as sending data in response to a promise, and custom drag drawing. We discuss those functions in "Providing Custom Drag Behavior" on page 32. In addition to the static functions just discussed, LDropArea provides a series of functions that are the heart of PowerPlant's implementation of drag and drop. Table 2.5 lists these functions.

Table 2.5 LDropArea functions

Function	Purpose
`PointInDropArea()`	return `true` if drag is inside the drop area (must be overridden)
`FocusDropArea()`	set up local coordinate system and clipping region for a drop area
`HiliteDropArea()`	highlight a drop area to indicate that it can accept the current Drag
`UnhiliteDropArea()`	remove highlight from a drop area
`EnterDropArea()`	called when drag enters a drop area
`LeaveDropArea()`	called when drag leaves a drop area
`DragIsAcceptable()`	return `true` when all items in the drag are acceptable

Function	Purpose
`ItemIsAcceptable()`	determine whether an individual item in the drag is acceptable
`InsideDropArea()`	called repeatedly while inside a drop area
`DoDragReceive()`	receive a drag
`ReceiveDragItem()`	receive an item in the drag

`PointInDropArea()` is a pure virtual function, and must be overridden. In addition, several other functions are empty, including `FocusDropArea()`, `InsideDropArea()`, `ItemIsAcceptable()`, and `ReceiveDragItem()`. You must provide this functionality in classes that derive from LDropArea.

In a typical implementation of drag and drop in PowerPlant, you override several of these functions (including the empty functions). We discuss which functions you commonly override in "Implementing Drag and Drop in PowerPlant" on page 24.

LDragAndDrop

LDragAndDrop is a mix-in class designed to add drag and drop features to a pane. It inherits from LDropArea and is a concrete implementation of that class. A pane that supports drag and drop should multiply inherit from LDragAndDrop, not LDropArea.

NOTE: A drag-savvy pane must override certain LDragAndDrop behaviors. We discuss which functions to override in "Implementing Drag and Drop in PowerPlant" on page 24.

LDragAndDrop adds one data member, `mPane`, to store a pointer to the pane associated with this drop area.

The LDragAndDrop class overrides three LDropArea functions, as shown in Table 2.6.

Table 2.6 LDragAndDrop functions

Function	Purpose
`PointInDropArea()`	determine if point is in the pane
`FocusDropArea()`	call `FocusDraw()` for the pane
`HiliteDropArea()`	highlight the pane frame

LDragAndDrop does not declare or define any new functions.

Now that you are familiar with the classes involved in drag and drop, let's examine how to actually implement drag and drop in a PowerPlant application.

Implementing Drag and Drop in PowerPlant

This section walks you through the process of implementing drag and drop in a typical PowerPlant application. As you know, this process involves creating a drag task, and having panes that can track and receive a drag. The tasks involved are:

- Looking for the Drag Manager
- Handling Clicks
- Identifying a Drag
- Creating a Drag Task
- Tracking a Drag
- Receiving a Drop
- Providing Custom Drag Behavior

Looking for the Drag Manager

Before using any routine that requires the presence of the Drag Manager, you should ensure that the Drag Manager is present. Simply call `LDropArea::DragAndDropIsPresent()` before any call that depends on the Drag Manager.

How to respond to the absence of the Drag Manager is, of course, a function of your application. If it absolutely requires drag and drop functionality, you should use weak import DragLib. This allows the application to launch even if the Drag Manager is not present. Then, at startup time, check for drag and drop. If it is not present, quit the application gracefully after alerting the user.

If your code is not dependent on drag and drop, you must implement alternative mechanisms, one for the drag manager, and one for when the drag manager is not present. You can do this by subclassing LDragAndDrop, if you carefully test for the presence of drag and drop before calling any Drag Manager or LDropArea routines, and you provide alternative code.

Handling Clicks

Handling clicks in a window that supports drag and drop is a two-step operation. First, you set an attribute for the window involved in the PPob resource for that window. Second, you override the `Click()` function inherited from LPane. Let's look at how these two tasks relate to each other.

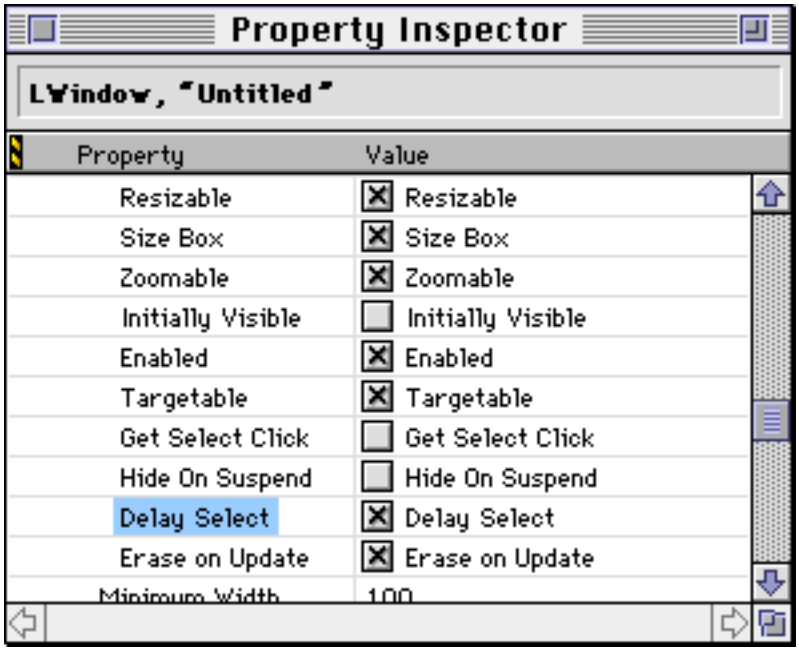
The drag and drop human interface guidelines are very specific about how a drag operation should appear to a user. The user should be able to click and drag data from an inactive window into another window, without the source window becoming activated.

NOTE: In the traditional human interface, selection highlighting disappears when a window becomes inactive. With drag and drop, the user still needs to see what objects are selected in an inactive window. This requires that you provide background highlighting as described in Apple's drag and drop documentation. The LTable and LTableView classes takes care of background highlighting automatically. TextEdit in the Toolbox provides background highlighting for both LTextEdit and LEditField.

PowerPlant provides support for clicking and dragging from a background window in the PPob resource through the **Delay Select**

option illustrated in Figure 2.2. This option corresponds to the `delaySelect` attribute of the LWindow class.

Figure 2.2 Getting the select click



When the `delaySelect` attribute is set, a click that would normally activate a window should be processed as if the window were already active. You accomplish that by overriding `Click()` for panes that support drag and drop.

As you know, `LPane::Click()` typically handles any click in a pane, view, or control. The default functionality usually provides everything you need. In the case of drag and drop, it does not.

The default code responds to clicks only when the `delaySelect` attribute is *not* set. Listing 2.1 shows the relevant code from LPane.

Listing 2.1 LPane::Click()

```
LPane::Click(SMouseDownEvent& inMouseDown)
{
  if (!inMouseDown.delaySelect) {
```

```
        // normal click handling code
    }
  }
}
```

You must override this function and replace the functionality. Otherwise, when you set the `delaySelect` attribute, your pane will not respond to clicks when it is in an inactive window. Replacement code for `Click()` might be organized like the code in Listing 2.2.

Listing 2.2 A `Click()` override

```
CMyClass::Click( SMouseDownEvent& inMouseDown)
{
  if ( inMouseDown.delaySelect )
  {
    // process the click for selection, etc.

    if( ::WaitMouseMoved(
                inMouseDown.macEvent.where))
    {
      // drag begins
    }
  }

} else
  {
    // Call inherited Click() for default behavior.
  }
}
```

Of course, the actual structure and functionality of your `Click()` routine depends upon your pane's contents and behavior. For example, in a graphics application you might check the location of the click against an existing selection. If the click is not on an already-selected item, you should deselect any existing selection, and select the clicked item. This same approach doesn't work for text, because you cannot select text with a single click.

After performing any preprocessing on the click, you then test for a drag, as described in "Identifying a Drag" on page 28. Notice you do this regardless of whether the click occurs in a foreground or background window, or changes a selection. This allows the user to perform a single-gesture selection and drag, as required in the drag and drop human interface. The process of handling a drag is described in the rest of this chapter.

If no drag begins, you simply exit the routine. In that case, control ultimately returns to the `LWindow::ClickInContent()` routine in PowerPlant, which activates the window for you.

For more information on topics relating to the drag and drop human interface, consult the drag and drop documentation.

Identifying a Drag

A drag begins when the user clicks and holds the mouse button down. Call the Mac OS routine `WaitMouseMoved()` to identify such an occurrence. The `WaitMouseMoved()` call returns a boolean value of `true` if a drag has begun. Listing 2.2 contains an example.

Creating a Drag Task

When a drag begins, you create an LDragTask object.

In PowerPlant, the LDragTask object manages the data necessary for the Drag Manager: the event that starts the drag, the drag reference, the data being dragged, and the drag region handle.

TIP: Before creating the LDragTask, call the window's `ApplyForeAndBackColors()` function to set the proper colors for the port and ensure that the drag highighting shows up correctly on non-white backgrounds, or for colored objects.

PowerPlant provides two alternative mechanisms for instantiating an LDragTask object. We will call them the simple approach and the flexible approach. Each approach has its own constructor function.

The simple approach

This technique is suitable for dragging a single item in a single flavor (data type). The constructor routine for the simple approach has several parameters. Here's the prototype for the constructor function.

```
LDragTask(const EventRecord &inEventRecord,
          const Rect &inItemRect,
          ItemReference inItemRef,
          FlavorType inFlavor,
          void *inDataPtr,
          Size inDataSize,
          FlavorFlags inFlags);
```

You provide the event record, the bounding rectangle, an item reference (typically the value 1), the flavor type, a pointer to the data, the size of the data, and any flavor flags.

When you call the constructor, PowerPlant takes over. The constructor function stores the event record, creates the necessary drag reference, adds the data to the drag in the flavor type you specify, and creates the drag region.

The drag region is always the bounds of the rectangle you provide in the second parameter. As a result, no matter what the underlying shape is, the drag outline that the user sees is rectangular.

The constructor also starts the drag by calling the Drag Manager's `TrackDrag()` function.

In the simple approach, the very act of instantiating the LDragTask object begins the drag operation automatically. However, the simple approach is suitable only for dragging one item in one flavor (data type).

The flexible approach

Use this approach if you want to drag multiple items, or if you want to provide multiple flavors for a single item. In practice, the flexible approach is much more useful. A well-designed application drags

data in a variety of flavors to increase the likelihood that the destination can understand the data. Remember, the destination may be any other process, not just the source application.

In addition, the flexible approach lets you set flags before starting a drag. For example, you can restrict the drag to the sender only.

The flexible approach uses a very simple LDragTask constructor.

```
LDragTask(const EventRecord& inEventRecord);
```

You provide the event record that starts the drag. The constructor function stores the event record, and creates the drag reference.

After that, you start the drag by calling the drag task's `DoDrag()` function. Here's the code for `LDragTask::DoDrag()`.

```
LDragTask::DoDrag()
{
  AddFlavors(mDragRef);
  MakeDragRegion(mDragRef, mDragRegion);
  ::TrackDrag(mDragRef, &mEventRecord,
              mDragRegion);
}
```

Notice that it calls two LDragTask functions, `AddFlavors()` and `MakeDragRegion()`. These are empty functions in LDragTask. You must subclass the LDragTask class and override these functions if you use the flexible approach.

In your definition of `AddFlavors()`, you write code to add data for multiple objects in multiple flavors. How you do that is dependent upon the data in your own application. Read the Drag Manager documentation for the steps to take and the calls to make to accomplish this task.

Similarly, in your definition of `MakeDragRegion()` you describe the outline of the various objects. Once again, consult the Drag Manager documentation for suggestions. You should also explore the `LDragTask::AddRectDragItem()` code to see how PowerPlant accomplishes the task for a simple rectangle.

Having provided these two functions, your initial work setting up the drag task and starting the drag is complete. The Drag Manager begins tracking the drag. During the process it calls your tracking handler and sends it messages.

Tracking a Drag

Every pane capable of receiving a drag should have a tracking handler. When the drag is within the bounds of the pane, the pane provides some visual feedback to the user indicating whether the drag can be received. For this to work, the pane must inherit from a class derived from LDragAndDrop.

In its simplest form, implementing drag tracking requires that you override two functions.

The first is `ItemIsAcceptable()`. In this function you use the Drag Manager to determine what data is in the drag. If you can accept it, you return `true`. Otherwise, you return `false`.

The second function you override is `HiliteDropArea()`. The default behavior of LDragAndDrop does *not* highlight a drop area. You must override `HiliteDropArea()` to provide the necessary highlighting.

If you want to get fancy, PowerPlant gives you the means to do so. There are several functions you may override if you wish to implement custom behavior. Table 2.7 lists the functions.

Table 2.7 Override for special tracking behavior

Function	Purpose
`UnhiliteDropArea()`	remove highlight from a drop area
`EnterDropArea()`	called when drag enters a drop area
`LeaveDropArea()`	called when drag leaves a drop area
`InsideDropArea()`	called repeatedly while inside a drop area

Override `UnhiliteDropArea()` if `HiliteDropArea()` does something other than call the Drag Manager's `ShowDragHilite()` routine.

Override `EnterDropArea()` and `LeaveDropArea()` if you want to implement special behavior when such an event occurs. The default behavior of these functions calls `HiliteDropArea()` and `UnhiliteDropArea()` respectively.

Override `InsideDropArea()` if the drag location inside the drop area affects what happens. For example, you may wish to indicate an insertion point in a text area.

Receiving a Drop

The final task you must perform is to receive the data in a drop. To accomplish this task, you must override the `ReceiveDragItem()` function.

In the standard implementation, the `DoDragReceive()` function walks through all the items in the drag, and calls `ReceiveDragItem()` for each one. You do not need to override `DoDragReceive()` in a typical application.

Once again, what data you can receive and how you receive it are matters that are application dependent and beyond the scope of PowerPlant. Read the Drag Manager documentation for the steps to take and for the calls to make to accomplish this task.

When a drag begins and ends in the same inactive window, PowerPlant activates the window automatically. This behavior is required by the drag and drop human interface guidelines. With that one exception, a successful drop does *not* activate the receiving window.

Providing Custom Drag Behavior

The Drag Manager gives you the option of replacing default Drag Manager behavior with special callback functions. You can use these advanced techniques to manage data delivery, drag feedback, and mouse and keyboard modifiers.

For example, the user may want to drag an item that contains a large amount of data. Rather than gathering up that data, duplicating it, and placing it in the drag, you can provide a "promise" to deliver the data to the destination. If the user aborts the drag you haven't wasted time or memory on an uncompleted operation. If the user completes the drag, the Drag Manager calls your custom callback function. This function fulfills your promise to deliver data.

Similarly, you may want to customize drag feedback. The Drag Manager draws a simple grey outline of the drag region. You may want to do something fancier, like drag an actual bitmap of the item. To do so, you must provide a custom callback function to handle drag drawing.

See the Drag Manager documentation for more information about the circumstances under which you may wish to replace the default Drag Manager behavior.

Notice that this custom behavior occurs on a per-drag basis. It has nothing to do with windows, panes, or applications. You provide the address of the callback function to the Drag Manager along with the drag reference for which you wish the callback to be used.

The LDropArea class has a variety of functions to assist you with these custom callback functions, as listed in Table 2.8.

Table 2.8 LDropArea custom callback support functions

Function	Purpose
`HandleDragSendData()`	send data callback routine
`HandleDragInput()`	user input callback routine
`HandleDragDrawing()`	drag drawing callback routine
`DoDragSendData()`	send data in response to a promise
`DoDragInput()`	modify mouse and modifier keys during a drag
`DoDragDrawing()`	do all drawing during a drag

The "handle" functions are the Drag Manager callback routines. They are static member functions. The code in each of the three "handle" functions is complete. You would not typically override that behavior. Each calls the corresponding "do" routine.

The code in each of the "do" routines is incomplete. In fact, all three "do" functions are empty. You must override these functions in a subclass and provide definitions.

Please note that the `LDropArea::InstallHandlers()` function only installs the tracking and receive handlers. The handlers to send data, control input, and perform drag drawing are *not* installed by PowerPlant.

If you want the Drag Manager to use these three PowerPlant handlers, you must call the appropriate Drag Manager routine to install the handler. The routines are: `SetDragSendProc()`, `SetDragInputProc()`, and `SetDragDrawingProc()`.

WARNING! When you set any of these three handlers for a particular drag, you must provide a pointer to the appropriate drop area as the `refcon` parameter. The PowerPlant handlers expect to find the LDropArea pointer in the `refcon` parameter when called by the Mac OS.

Summary of Drag and Drop in PowerPlant

Drag and drop is a powerful tool. Users love the ease of use and intuitive freedom it provides for copying and pasting data.

However, implementing drag and drop is not always simple. PowerPlant eliminates much of the pain of basic implementation. It does not eliminate all of the complexity. As you learned, in many cases you must still prepare the data in various flavors, and add that data to the drag. You must also handle receiving the data. This can be a non-trivial task.

However, PowerPlant gives you a robust framework on which you can hang your drag and drop code. PowerPlant takes care of dispatch and control. You provide the functions to determine if a drag can be received, to receive the items in the drag, and (typically) to add data to the drag and create a drag region.

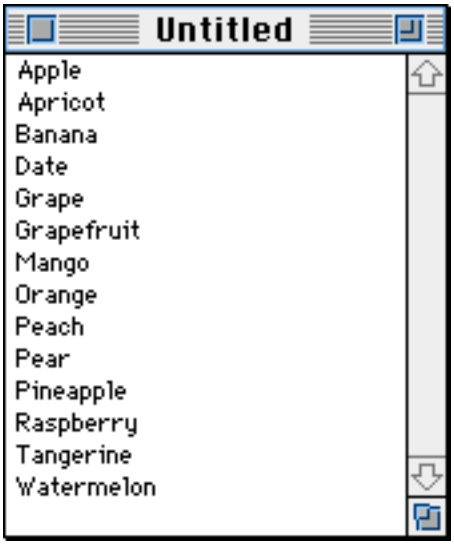
PowerPlant also provides a variety of functions you can use to customize your drag and drop behavior. You can implement a full range of features using descendants of the LDropArea class.

Code Exercise for Drag and Drop

In this exercise you implement simple drag and drop in an application named "DragAndDrop." The purpose of this exercise is to give you experience with drag and drop in PowerPlant—the functions you override and the tasks you perform. This exercise is not intended as a tutorial on the Drag Manager.

The vehicle for drag and drop is a small window containing an instance of the PowerPlant LTable class. Figure 2.3 shows the window. It contains a single-column table that lists a variety of fruits.

Figure 2.3 The DragAndDrop window



The general tasks you must perform to support drag and drop in a PowerPlant application are:

- Handle clicks
- Identify a drag
- Create a drag task
- Track the drag
- Receive the drop

In this exercise you write code to accomplish each of these tasks.

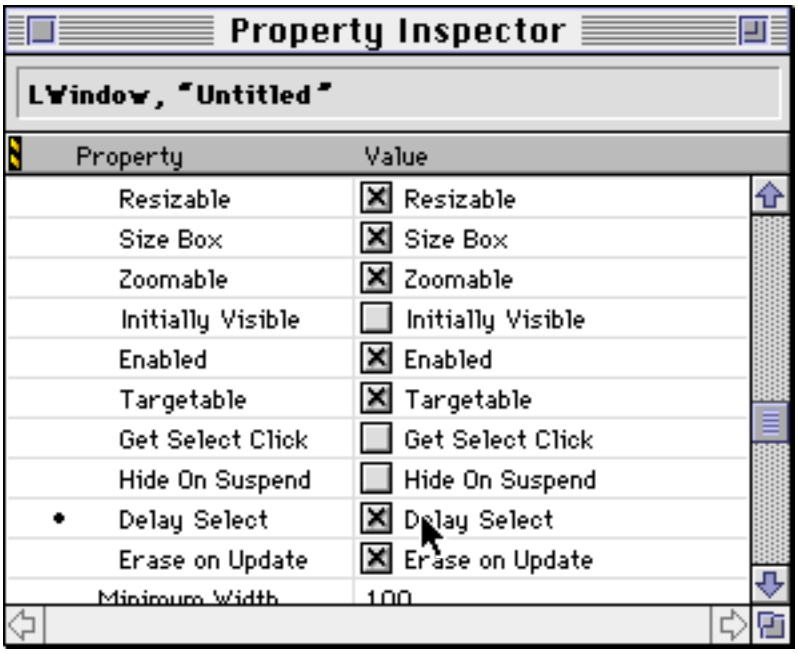
1. Support drag and drop in an inactive window.

`Drag Window resource` `DragAndDrop.ppob`

In this step you modify the PPob resource to enable the `delaySelect` attribute of the window. Open the DragAndDrop.ppob file in Constructor, open the Drag Window resource, and open the property inspector for the drag window.

Click on the **Delay Select** check box to turn it on. With this option active, PowerPlant will delay activating an inactive window. This gives you an opportunity to handle the click for drag and drop.

Figure 2.4 Turning on Delay Select



When you are through, save your changes and close the file. You can quit Constructor is you wish.

2. Handle clicks.

`Click()` `CDragAndDropTable.cp`

In order to support drag and drop, when a click occurs you must explicitly test for the `delaySelect` attribute, and the presence of drag and drop. If both conditions exist, then you can process the click.

The existing code has an empty `if` test. Test the `delaySelect` field of the `inMouseDown` parameter, and call `DragAndDropIsPresent()`. The code controlled by the if statement is provided for you. It does the required bookkeeping and calls `ClickSelf()`.

```
if ( inMouseDown.delaySelect
     && DragAndDropIsPresent() ) {
```

With these two steps you have accomplished the first principal task, handling clicks properly. Next, you identify the beginning of a drag.

3. Identify a drag operation.

`ClickCell()` `CDragAndDropTable.cp`

The `LTable::ClickSelf()` function identifies which cell in the table has been clicked, and calls `ClickCell()`. CDragAndDropTable overrides `ClickCell()` to support drag and drop. This is where you detect an incipient drag.

The existing code has an empty `if` test. You must test for the presence of the drag manager, and whether a drag has commenced.

To test whether a drag has commenced, call the Mac OS Toolbox function `WaitMouseMoved()`. Pass `inMouseDown.macEvent.where` as the parameter to `WaitMouseMoved()`.

```
if (  DragAndDropIsPresent()
      && ::WaitMouseMoved(
                  inMouseDown.macEvent.where ) ) {
```

The code inside this `if` statement executes if and only if the user is dragging and the Drag Manager is present. You have successfully accomplished the second principal task, detecting a drag. Next, you create a drag task.

4. Create a drag task

`ClickCell()` `CDragAndDropTable.cp`

The code you write in this step goes inside the `if` statement you modified in the previous step. If the user is dragging, you create a drag task.

The existing code manipulates table information and sets some local variables. It gets the cell, the cell frame in a `Rect`, and the cell data in an `Str255`. After that, you create a drag task.

This code supports dragging a single item in a single flavor, so you can implement the "simple" approach to a drag task. This constructor requires that you pass the event record involved, the bounds of the drag region, an item reference, the flavor of data, the address of the data, the length of the data, and flavor flags.

```
GetCellData( theCell, theString );
// Create the drag task.
LDragTask theDragTask( inMouseDown.macEvent,
        theCellFrame, 1, 'TEXT',
```

```
                &theString[1],
                StrLength(theString), 0 );
```

Remember that creating a drag task in this way also starts the drag running automatically. There is no need to call `DoDrag()`.

After this code, the existing code performs one more step required to conform to the human interface guidelines for the Drag Manager. The code checks to see if the drag ended up in the trash. Here's the relevant code. You don't have to type this in, it already exists.

```
if ( UDragAndDropUtils::DroppedInTrash(
            theDragTask.GetDragReference() ) ) {
```

If the drag ends up in the trash, the code removes the item from the table. You have now completed the third principal task required to support drag and drop. You have created the drag task and started the drag running. In the next three steps you provide the support required to track a drag.

TIP: You may find the UDragAndDropUtils class very useful in your own coding. It has static functions to test for a drop in the trash, whether the user has the option key down, and whether the drop is in the same window it started in. These are very useful housekeeping routines.

5. Determine if the drag is acceptable.

`ItemIsAcceptable()` `CDragAndDropTable.cp`

The `ItemIsAcceptable()` function is a utility function that PowerPlant calls when it needs to know if a drop can be received by a particular drop area. To properly support drag and drop, you must define this function.

The table is capable of receiving a text item. You should call the Drag Manager's `GetFlavorFlags()` function and determine if the item has the TEXT flavor. You should also test to ensure that the table is enabled (this is an LPane function, `IsEnabled()`).

```
FlavorFlags theFlags;
return IsEnabled()
```

```
        && (::GetFlavorFlags( inDragRef,
        inItemRef, 'TEXT', &theFlags ) == noErr);
```

6. Provide drag feedback.

`HiliteDropArea()` `CDragAndDropTable.cp`

PowerPlant calls this function when it is necessary to highlight a drop area. This provides the user with visual feedback that the drop is acceptable in the drop area.

In this case, the drop area is the window's content area. This function should get the local frame bounds, convert it into a region, call the Drag Manager's `ShowDragHilite()` function, and dispose of the region.

```
// Get the frame rect.
Rect theRect;
CalcLocalFrameRect( theRect );

// Show the drag hilite in the drop area.
RgnHandle theRgnH = ::NewRgn();
::RectRgn( theRgnH, &theRect );
::ShowDragHilite( inDragRef, theRgnH, true );
::DisposeRgn( theRgnH );
```

Because you use the `ShowDragHilite()` function, you do not need to override `UnhiliteDropArea()`.

7. Provide insertion point feedback.

`InsideDropArea()` `CDragAndDropTable.cp`

In the DragAndDrop application, the drop area is the table, and the table occupies the content area of the window. To be very friendly, there should also be an insertion point to clearly demonstrate where the drop is going to end up in the list.

To make this work, you must track the drag while it moves around inside the drop area. While a drag moves inside a drop area, PowerPlant repeatedly calls `InsideDropArea()`.

The existing code does all the setup work. This application uses the `mDropRow` data member to track where in the table the drop should go. The existing code calls the inherited `InsideDropArea()` func-

tion, gets the mouse location for the drag, and determines what row is involved.

If it's a new row, it erases the previous dividing line (insertion point). After that, you should draw a new dividing line.

Save `theRow` in the `mDropRow` data member. Then call `DrawDividingLine()`.

```
mDropRow = theRow;
DrawDividingLine( mDropRow );
```

The code for determining the row and drawing the dividing line is provided for you. Feel free to examine it.

NOTE: The CDragAndDropTable class also overrides `EnterDropArea()` and `LeaveDropArea()`. These overrides simply invalidate the `mDropRow` value so that a new insertion point is drawn when the drag changes cells.

With this step you have completed the support required for tracking a drag. All that remains is receiving a drop.

8. Receive the drag item.

`ReceiveDragItem()` `CDragAndDropTable.cp`

As you know, the purpose of this function is to retrieve data from the drop. The code in this function is provided for you. It has more to do with manipulating data in an LTable than anything else.

Examine the code, and understand the tasks it performs. First, it gets the data in the drag item by calling the Drag Manager's `GetFlavorData()` function. Then it gets the size of the data by calling `GetFlavorDataSize()`.

After that, the code determines if the drag is a move or a copy operation. It is a move operation if the drag is in the sender window, and the option key is not down. Otherwise it is a copy operation. Notice

that the code uses the UDragAndDropUtils member functions for both tests.

If it is a move operation, the code removes the data from one location in the table and inserts it in another.

If it is a copy operation, the code inserts the data in the proper location in the table.

9. Build and run the application.

When the application builds successfully and runs, the window shown in Figure 2.3 on page 35 appears with the list of fruits.

Click and drag an entry in the list to another location in the list. Notice that you can select and drag an item with a single gesture. This is an important feature of the human interface.

While a drag is underway, observe the insertion point feedback (the black line between entries). The insertion point is the work of the `InsideDropArea()` function. As the drag moves, the insertion point follows along.

Notice that there is no drop area highlighting (the border around the content area of the window). Drop area highlighting only appears after a drag has left the sender window. To see it, drag the item out of the window, and then back into the window. When you reenter the drop area, the drop area highlighting appears indicating that this is an acceptable drop. The highlight is from the `HiliteDropArea()` function.

A drag within a single container can be a move or a copy. Hold down the option key while you begin a drag, or when you drop an item in the same table as you started. You don't have to hold the option key down for the entire drag, just at the beginning or the end. If the option key was down at either of those moments, the original remains and a copy is placed at the drop location when you drop the item.

Drag an item to the trash and observe what happens. The item should disappear from the table. Look in the trash, and you'll see a text clipping. You can drag that item back out of the trash and into a table if you wish.

Now, make a new window and drag items back and forth between windows. In particular, drag an item from the inactive window into

the active window. A drag from an inactive window or a drop into an inactive window does not cause that window to become active. This is another important feature of the human interface. Notice also that the inactive window shows background highlighting of the currently selected table cell in that window.

Finally, drag an item from a window and drop it on the Finder's desktop. The item appears on the desktop as a text clipping. Drag the text clipping into a table. The text appears as a new entry. Drag and drop works across applications!

Launch a text processor that supports drag and drop, like SimpleText. Type a phrase into SimpleText. Then drag it into the DragAndDrop table. The text transfers from one application to another with no difficulty. Drag something from the table into SimpleText. It works both ways.

Play around with drag and drop. It makes data transfer so simple that you might wonder why it hasn't always been this easy. When you are through playing, quit the application.

PowerPlant gives you a lot of drag and drop functionality for free. PowerPlant takes care of most of the housekeeping details, dispatching control to the proper functions when necessary. You override and define a few, relatively simple hook functions such as `ItemIsAcceptable()`, `HiliteDropArea()`, and so forth.

As always, there is room for further exploration. You could implement drag and drop for pictures. Follow the same steps as you did in this exercise, but use PICT data instead of text.

For a greater challenge, you might add multiple flavors of the same data to a drag (such as both TEXT and PICT). Or you might add multiple items to a single drag. You might create a drag region that outlines each individual item in a multiple-item drag, rather than the encompassing rectangle. See "The flexible approach" on page 29 for ideas on how to go about this.

Finally, for a real challenge, you could implement custom routines for managing a drag operation. For example, you could drag around a bitmap image of the item being moved, rather than a sim-

ple outline. See “Providing Custom Drag Behavior” on page 32 for hints and clues.

Good luck, and happy dragging.

3

Threads in PowerPlant

This chapter discusses how to use the PowerPlant threads classes. You might use these classes in a PowerPlant application, or in non-PowerPlant program.

Introduction to Threads in PowerPlant

A task may require a substantial period of time to complete. In the context of a computer program, anything over one second is a substantial period of time. If you seize control of the computer for one second or more without allowing the user to perform typical actions such as choosing a menu item, your software can be seen as unfriendly or slow.

How can you undertake a computationally-intensive task without causing the machine to appear sluggish or non-responsive? The solution is to have time-consuming tasks running simultaneously with other operations, such as managing the user interface. One way to accomplish this goal is to run each task in concurrent but separate threads of execution.

Threads are sometimes called lightweight processes. They are like processes in that they represent the execution of some program code, and because they provide a mechanism for multitasking. They are lightweight in the sense that they don't require as much state information as normal processes. Therefore it is relatively cheap to create, destroy, and switch between threads.

The Thread Manager is an implementation of the threads concept for Mac OS computers. The Thread Manager provides a way for applications to divide their work into discrete, independent subtasks.

The Thread Manager switches between the various threads. In essence, each application gets its own multitasking environment.

The topics in this chapter include:

- The Thread Strategy—PowerPlant's approach to threads
- Thread Classes—a detailed look at the PowerPlant classes involved in thread support
- Implementing Threads in PowerPlant—how to implement simple threads
- Data Coherency—techniques for ensuring the reliability of shared data
- Asynchronous Operations—advanced thread operations
- Summary of Threads in PowerPlant

There is a lot of material in this chapter, some of it very deep and involved. Do not let that frighten you. A straightforward implementation of threads in a PowerPlant application is actually quite simple. The code exercise at the end of the chapter demonstrates how easy it is to implement useful, custom threads with PowerPlant.

This chapter does not teach you the intricacies of the Thread Manager. For more information, consult *Inside Macintosh: Thread Manager*. The article "Concurrent Programming with the Thread Manager" in issue 17 of develop magazine may be of interest, although it might be slightly outdated because the Thread Manager no longer supports preemptive threads.

The Thread Strategy

The Mac OS Thread Manager implements threads at a basic level. PowerPlant's approach enhances threads in two ways.

First, the PowerPlant classes provide a wrapper for the Thread Manager, insulating you from the Toolbox. This can make simple or typical threads very easy to implement.

Second, PowerPlant adds significant utility to the Thread Manager in the following ways:

- Thread States—PowerPlant adds new thread states
- Semaphores—PowerPlant provides direct support for semaphores
- Inter-Thread Communication—PowerPlant provides for direct communication between threads

Thread States

The Thread Manager defines three possible thread states, current, ready and stopped. PowerPlant implements six thread states. In PowerPlant, a thread is always in one (and only one) of the states described in Table 3.1.

Table 3.1 Possible thread states

State	The thread is…
current	in control of the CPU
ready	waiting its turn for the CPU
suspended	not eligible for CPU time
sleeping	sleeping for a specified time
waiting	waiting for a semaphore to clear
blocked	waiting for an asynchronous I/O call to complete

The current thread is the thread in control of the CPU. There is always one (and only one) current thread. When the current thread transfers control of the CPU to another thread, it goes into the ready state. Threads move between the current and ready states through a call to `LThread::Yield()`. The current state in PowerPlant corresponds to the Thread Manager's current state.

A ready thread is a thread that is not current, but is otherwise eligible for execution. In other words, only a ready thread may become the current thread. The Thread Manager's scheduling algorithm schedules ready threads on a round-robin basis. This state corresponds to the Thread Manager's ready state.

A suspended thread is ineligible for CPU time. Call `LThread::Suspend()` to suspend a thread. Call `LThread::Resume()` to return the thread to the ready state. The suspended state corresponds to the Thread Manager's stopped state.

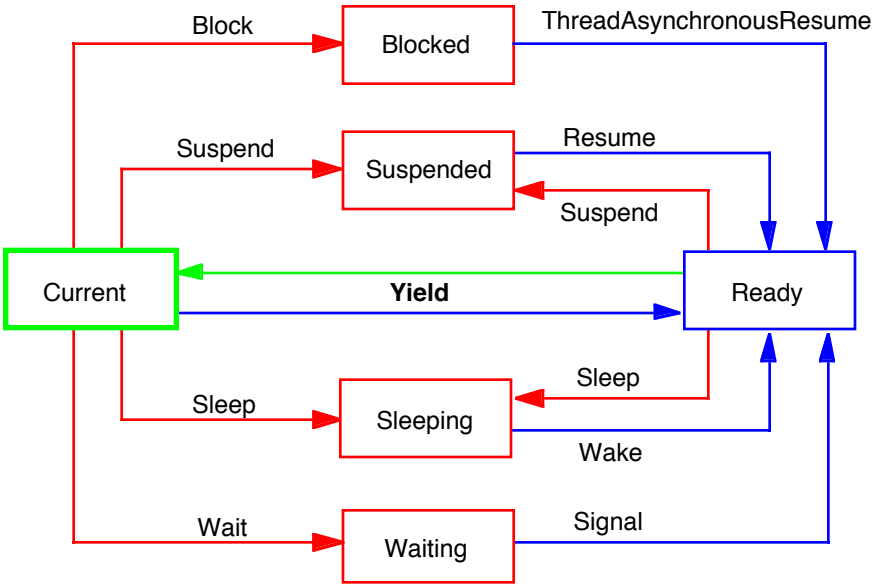
A sleeping thread is ineligible for CPU time for a certain period of time. Call `LThread::Sleep()` to put a thread to sleep. A sleeping thread returns to the ready state when the sleep time expires, or if you call `LThread::Wake()`.

A waiting thread is ineligible for CPU time while a semaphore (flag) indicates it cannot run. When the semaphore is cleared, the thread returns to the ready state. Call `LSemaphore::Wait()` to set a semaphore. Call `LSemaphore::Signal()` to clear a semaphore. See "Using Semaphores" on page 74 for more information.

A blocked thread is ineligible for CPU time. It is waiting for an asynchronous I/O call to complete. Call `LThread::Block()` to block a thread after making an asynchronous I/O call. Calling `LThread::ThreadAsynchronousResume()` unblocks the thread. See "Asynchronous Operations" on page 81 for more information on blocking and asynchronous I/O.

Figure 3.1 contains a simplified state transition diagram for thread objects.

Figure 3.1 State transition diagram for thread objects



Transitions from stopped states make the thread ready, not current. A thread becomes current only by a yield from the current thread.

Semaphores

When multiple threads access the same data, big trouble can result if one thread changes data that another thread is using. Keeping shared data access under control is a major issue in concurrency.

The Thread Manager in the Toolbox has a feature for preventing two threads from accessing the same data simultaneously. `ThreadBeginCritical()` turns off thread scheduling, so any yield has no effect. `ThreadEndCritical()` turns thread scheduling back on. In effect, these calls allow you to temporarily convert your application into a single-threaded program.

This mechanism is inadequate in more complex situations. You may want to yield to allow your application to be responsive, and yet still preserve the integrity of the data on which you are operating.

PowerPlant allows you to easily create semaphores that prevent other threads from modifying data until the semaphore is cleared. The classes involved are LSemaphore, LEventSemaphore, and LMutexSemaphore. The "mutex" is short for mutually exclusive. LMutexSemaphore defines a particular kind of semaphore where two or more threads are mutually exclusive. One and only one mutually exclusive thread may be ready. All others are put in the wait state.

Inter-Thread Communication

You may want two threads to be able to communicate directly with each other. A typical situation is that one thread produces data that another thread uses. There is no mechanism in the Thread Manager to implement inter-thread communication.

PowerPlant has such a mechanism. You store shared data in a class derived from LLink. You then create a queue of data with LSharedQueue (derived from LQueue). You must have a queue of some sort, because you cannot expect the producing thread to generate data at the same rate that the consuming thread uses it. The LSharedQueue class also inherits from LMutexSemaphore so that you can ensure that the data production and consumption threads do not interfere with each other.

When you create the threads involved, you provide the single queue that each uses for data transfer. The producer puts data in the queue, and the consumer removes data from the queue as necessary.

Thread Classes

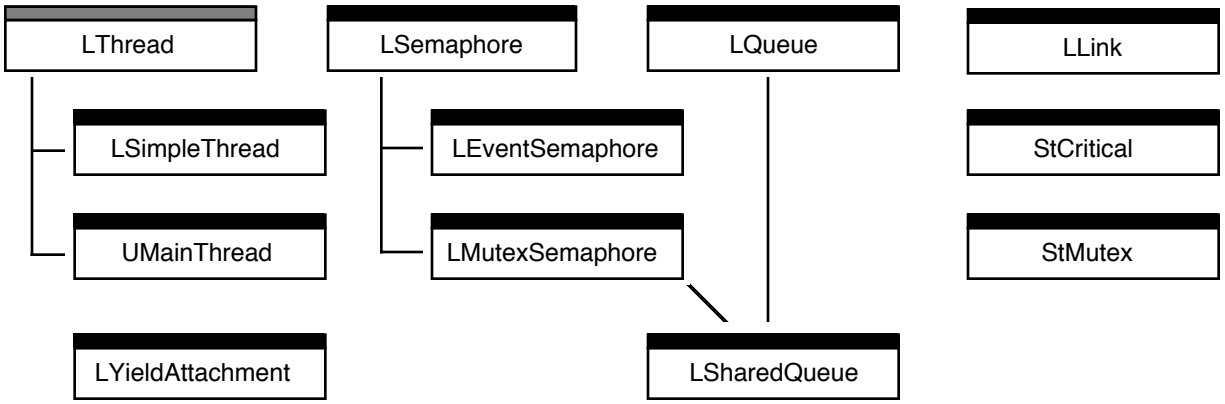
PowerPlant has several classes related to thread support:

- LThread
- LSimpleThread
- UMainThread
- LYieldAttachment
- LSemaphore
- LEventSemaphore

- LMutexSemaphore
- StMutex
- StCritical
- LLink
- LQueue
- LSharedQueue

One group of classes relates directly to threads. Another group implements semaphores. A third groups implements thread linking and data queues. Figure 3.2 illustrates the class hierarchies.

Figure 3.2 The PowerPlant thread-related classes



The grey bar on the LThread class indicates that LThread is an abstract class. Taken together, these classes form an independent module in PowerPlant. You can use these classes in non-PowerPlant code if you wish.

LThread

The LThread class forms the basis for thread support in PowerPlant. This is a complex class with many data members and member functions. This discussion covers those you are most likely to encounter directly. Use the *PowerPlant Reference* for complete information.

NOTE: The LThread class includes support for preemptive threads, but you should not create preemptive threads. Earlier versions of the Thread Manager had preemptive thread scheduling as well as cooperative scheduling on 68K Macintosh computers. Copland and future operating systems will not support preemptive scheduling within an application. As a result, the Thread Manager no longer supports preemptive scheduling.

The discussion of LThread covers the following topics:

- LThread attributes—LThread data members
- LThread Behaviors—LThread member functions
- The Run() function—special considerations surrounding a thread's `Run()` function

LThread attributes

LThread has several data members you may find useful. Table 3.2 lists these members.

Table 3.2 Some LThread data members

Member	Stores
`sThread`	the current thread
`sMainThread`	the main thread
`sThreadQueue`	a queue of all threads
`sPSN`	the process serial number for the application
`mThread`	the thread ID
`mState`	the thread's state (current, ready, etc.)
`mResult`	result of the thread
`mNextOfKin`	thread to be notified when this thread dies
`mSemaphore`	semaphore that the thread waits for (if any)
`mError`	stores any error that occurs while waiting

The “s” data members are static, and therefore are class variables. There is one current thread, one main thread, one thread queue, and one process serial number for the application. Those values are shared by all threads.

The `mNextOfKin` data member deserves special mention. This data member stores a pointer to another LThread object. When the thread completes its function and is about to be deleted, the next of kin is notified and can retrieve the result generated by the dying thread. See “The Run() function” on page 55 for more information about a thread’s result and next of kin.

LThread Behaviors

LThread has a series of static member functions that are always available, even if you don’t happen to have a thread object handy. Table 3.3 lists several static member functions.

Table 3.3 Some LThread static member functions

Function	Purpose
`AllocateThreads()`	preallocate some threads
`GetFreeThreads()`	return number of unused preallocated threads
`GetCurrentThread()`	return current thread
`GetMainThread()`	return main thread
`CountReadyThreads()`	return number of ready threads
`InMainThread()`	return `true` if current thread is the main thread
`DoForEach()`	iterate over all threads and perform a task
`Yield()`	surrender time to other threads

Function	Purpose
`EnterCritical()`	turn off thread scheduling so yields are ignored
`ExitCritical()`	turn on thread scheduling

The `Yield()` function is safe to call, even if the Thread Manager is not present. This call causes the current thread to switch to the ready state. The Thread Manager then switches another thread into the current state and gives it control of the processor.

You can use the `DoForEach()` function to walk the list of threads for any purpose. You specify an `LThreadIterator` procedure to and pass any associated data required by that function.

The remaining functions are self-explanatory. Remember, each of these is static. You can use them at any time.

In addition to `Yield()`, LThread includes other non-static functions to change or query a thread's state, as listed in Table 3.4.

Table 3.4 LThread state functions

Function	Purpose
`Resume()`	make the thread ready, yield
`Suspend()`	suspend the thread, yield
`Sleep()`	put the thread in the sleep state, yield
`Wake()`	make a sleeping thread ready, yield
`Block()`	put a thread in the blocked state
`IsCurrent()`	return `true` if the thread is current

See the state transition diagram in Figure 3.1 for a graphic representation of the effect of these calls.

The `Resume()` function is very important. Threads are created in the suspended state. No thread will operate unless you call `Re-`

sume() after creating the thread. Calling Resume() results in a yield.

You specify the number of milliseconds in the Sleep() call. If you do not specify a value, the thread is put to sleep indefinitely. Calling Sleep() also results in a yield.

There are a few more vital functions in LThread, listed in Table 3.5.

Table 3.5 Some vital LThread functions

Function	Purpose
LThread()	constructor
SwapContext()	called each time a thread becomes current or not current
Run()	perform the thread's designed task
SetResult()	store value in mResult
GetResult()	retrieve value from mResult
SetNextOfKin()	set the next of kin thread
DeleteThread()	destroy a thread

LThread is an abstract class, but you call the constructor function from derived thread class constructors. "Creating Threads" on page 65. You can use default values for the constructor parameters.

You can use the SwapContext() function to perform housekeeping chores necessary when a thread comes into or goes out of current status. "Context Switching" on page 72.

The remaining vital functions are all deeply intertwined with the Run() function.

The Run() function

Run() is declared as a pure virtual function in LThread. You must override and define this function in any derived class. The Run()

function should perform the task for which the thread is designed. You never actually call the `Run()` function. The Thread Manager executes this function when the thread becomes current. When the `Run()` function completes, it returns a void pointer to data. That value is automatically placed in the `mResult` data member.

However, the result is of limited use. The sequence of events when a thread completes is as follows:

- `Run()` returns a value (a void pointer).
- `DeleteThread()` is called automatically for the thread.
- `DeleteThread()` calls `SetResult()` to store the return from `Run()` in `mResult`.
- `DeleteThread()` prepares to destroy the thread.
- The next of kin is notified that the thread is about to die.
- The thread is destroyed.

The only time the value in `mResult` is useful is to the next of kin. When notified that a thread is about to die, the next of kin can send the thread (which is near death but not yet dead) a `GetResult()` message. If you wish to use this feature, call the `SetNextOfKin()` function after creating a thread. The next of kin will be notified just before the thread is deleted.

If the thread generates data that you wish to survive beyond the life of the thread's `Run()` function, and you do not use the next of kin feature, you must store that data in some variable or object that persists outside the scope of the thread object.

If the thread has completely run its course, there is no need to destroy the thread. PowerPlant takes care of that for you.

A thread may serve a short-term purpose. For example, you might spawn a thread to perform a single calculation. A thread may also last as long as the application runs. You might have a thread running continuously to look for a particular event, perhaps an incoming message from a network. A thread exists as long as the `Run()` function does not return, and you do not call `DeleteThread()`.

WARNING! If you wish to destroy a thread before it has completed its `Run()` function, you must use `DeleteThread()`. Do not use `operator delete` to destroy a thread. There is a good deal of cleanup work required to properly dispose of a thread.

LSimpleThread

LSimpleThread is a concrete subclass of LThread. It has two additional data members:

- `mProc`—a `ThreadProc` pointer to the function you want to execute when the thread runs
- `mArg`—a pointer to data you want passed to the ThreadProc

A `ThreadProc` function has the following prototype:

```
void MyThreadProc(LThread& thread, void* arg);
```

When you instantiate an LSimpleThread object, you specify the `ThreadProc` pointer and the `void` pointer to data. The LSimpleThread implementation of `Run()` simply calls the `ThreadProc` function and passes the argument. Here's the code:

```
void* LSimpleThread::Run()
{
  (*mProc)(*this, mArg);

  return (mResult);
}
```

The `mArg` parameter, because it is a void pointer, can hold any kind of pointer. This includes pointers to objects. For example, you might pass in a pointer to an LWindow object so the thread can manage a window.

Note that `LSimpleThread::Run()` returns the value of `mResult` as a void pointer. This assumes that the `ThreadProc` function sets the value of `mResult` before returning. This value is `NULL` by default, and may remain `NULL` if you never use it.

LSimpleThread lets you implement threads with an absolute minimum of pain. You don't have to declare or define any thread-related classes. You write a function that matches the `ThreadProc` prototype. Then you instantiate an LSimpleThread to run that function. In your `ThreadProc` function you should call `LThread::Yield()` regularly. If there are data synchronization issues you must, of course, pay attention to them. We discuss many of these problems in detail later in this chapter. Otherwise, you have everything necessary to implement threads!

WARNING! Don't forget to call `Resume()` after you create the LSimpleThread.

UMainThread

UMainThread is a concrete implementation of LThread. You must create a UMainThread object (or derivative) when your application launches, and before you create any other thread objects.

This class declares no new data members or member functions. UMainThread's implementation of `Run()` is empty. What you're really doing is creating a PowerPlant thread object that corresponds to the Thread Manager's thread for your process. The main thread typically includes the main event loop.

When you create the UMainThread object, it is already running. You do not need to call `Resume()`. The `UMainThread::Run()` function should never be called or entered.

See also: "Initializing Threads" on page 63.

LYieldAttachment

The LYieldAttachment class derives from LAttachment. You should not have to override this class in typical circumstances. This attachment simply calls `LThread::Yield()`.

There is one data member of interest, `mQuantum`. You provide the value of `mQuantum` when you create the attachment object. This

value should be in ticks (60ths of a second). It controls how the attachment yields. The default value is -1.

If `mQuantum` is negative, the attachment yields once. If `mQuantum` has a positive value, the attachment sits in a loop until that many ticks have passed, calling `Yield()` repeatedly.

In typical use, you specify -1 as the value for `mQuantum`. You then attach an LYieldAttachment object to the application object. This ensures that the main thread (usually a UMainThread object) yields repeatedly, because the attachment's `ExecuteSelf()` function will be called for every event.

Although commonly attached to the application object for the main thread, you can use LYieldAttachment with any host.

LSemaphore

LSemaphore is the base class for PowerPlant's implementation of semaphores. Most of the data members and functions in LSemaphore are internal to PowerPlant. You won't have to concern yourself with their operation.

Each semaphore object has an `mThreads` data member. Functionally, this is a list of all threads waiting for the semaphore to clear before accessing the flagged data.

The only two data members you will typically concern yourself with are:

- `Wait()`—raise the semaphore to protect data
- `Signal()`—lower the semaphore when data access complete

In typical circumstances that's what happens. In fact, the sempahore keeps a count of the waits and signals. Calling `Wait()` increments the counter, and calling `Signal()` decrements the counter. The state of the counter determines whether the data is protected or not.

You may specify the number of milliseconds you are willing to wait in the call to `Wait()`. If you do not, the default value of the parameter is that you will wait forever.

See also: "Using Semaphores" on page 74.

LEventSemaphore

LEventSemaphore is a subclass of LSemaphore. It has two features that distinguish it from LSemaphore.

LEventSemaphore overrides the `Signal()` function. The `LEvent-Semaphore::Signal()` function simultaneously releases *all* threads waiting on this semaphore.

In addition, LEventSemaphore declares a new function, `Reset()`. This function is guaranteed to raise the semaphore so that no thread can access the flagged data until the next call to `Signal()`.

LMutexSemaphore

LMutexSemaphore implements a mutually exclusive semaphore. If a mutually exclusive semaphore is raised, only one thread may access the flagged data. When the semaphore is lowered, if there are waiting threads, only one waiting thread is returned to the ready state.

LMutexSemaphore declares a new data member, `mOwner`, the thread that currently owns the semaphore. Only the owner may access the data protected by the semaphore.

LMutexSemaphore also overrides both `Wait()` and `Signal()` to implement the proper behavior.

See also: "Using Semaphores" on page 74.

StMutex

StMutex is a stack-based utility class to implement an exception-safe and simple mutual exclusion semaphore. You create the semaphore first. You pass the semaphore to the constructor. The constructor calls `Wait()`. The destructor calls `Signal()`. If an exception is thrown while the semaphore is raised, the destructor is still called and the semaphore is lowered correctly.

StCritical

StCritical is a stack-based utility class to implement an exception-safe and simple method for using the Thread Manager's `ThreadBeginCritical()` and `ThreadEndCritical()` functions. When you declare an StCritical object, the constructor calls `ThreadBeginCritical()`. The destructor calls `ThreadEndCritical()`. If an exception is thrown during the critical segment, the destructor is still called and the critical exclusion is released correctly.

LLink

The LLink class is an extraordinarily simple implementation of a linked list. The class declares one data member, `mLink`, which is a pointer to the next LLink object.

There are two substantive member functions, `SetLink()` and `GetLink()`. These serve as accessors for the `mLink` data member.

You typically would not use this class directly. The true purpose of LLink is to serve as a base class for custom subclasses. A typical subclass of LLink adds data members that store data you wish to pass between threads in an LQueue object.

LQueue

LQueue implements full linked list behavior for a list of LLink objects. It has the features you would expect to find in a class that manages a linked list.

There are three data members, as shown in Table 3.6.

Table 3.6 LQueue data members

Data member	Stores
`mFirst`	first element in the linked list
`mLast`	last element in the linked list
`mSize`	number of elements in the linked list

Every element in an LQueue is an LLink object. The member functions also work on LLink objects. Table 3.7 lists the LQueue member functions.

Table 3.7 LQueue member functions

Function	Purpose
`GetSize()`	returns number of items in the list
`IsEmpty()`	returns true if the list is empty
`NextPut()`	add an element to the end of the list
`NextGet()`	get and remove the first element in the list
`Remove()`	remove an element from the list
`DoForEach()`	walk through the list and perform a task

LQueue implements a first in, first out (FIFO) queue. You may only add items to the end of the list, and you may only get items from the head of the list. Typically you would not call `Remove()` directly, but you can use it to remove any arbitrary item from the list.

TIP: There is nothing that limits the use of LLink and LQueue to threads. You can use this combination of classes to implement a simple FIFO queue in any context.

LSharedQueue

LSharedQueue inherits from both LQueue and LMutexSemaphore. This class combines a queue with a semaphore so that you can protect the data in the queue. The design purpose of LSharedQueue is to allow two or more threads to share a common data queue.

As one thread adds items to the queue, other threads that read data from the queue are locked out to protect the integrity of the queue. Similarly, if a thread is retrieving an item from the queue, other threads are prevented from modifying the queue.

LSharedQueue declares a new member function, `Next()`. When retrieving data from an LSharedQueue, you should call `Next()` and not call `NextGet()`. The `Next()` function does the semaphore testing, and if the queue is available calls `NextGet()`. The `Next()` function also allows you to specify how long you are willing to wait for the data, because your access to the queue may be blocked.

See also: "Inter-Thread Communication" on page 78.

Implementing Threads in PowerPlant

This section discusses the tasks you must perform to implement simple threads in PowerPlant code. The topics discussed include:

- Initializing Threads—setting up for thread operations
- Creating Threads—instantiating individual threads
- Running a Thread—getting a thread started
- Modifying Thread State—how to change thread state
- Deleting Threads—what to do when a thread is finished

Initializing Threads

To use threads in PowerPlant you must do two things: ensure that the Thread Manager is available, and create a main thread.

Determine if the Thread Manager is present

Before creating thread objects, ensure that the Thread Manager is available. PowerPlant sets the UEnvironment `sFeatures` data member at startup. To determine if the Thread Manager is available, simply call `UEnvironment::HasFeature()`, as shown here.

```
if (UEnvironment::HasFeature
                        (env_HasThreadsManager))
{
  // Thread Manager available
}
```

TIP: If you check for the Thread Manager at application startup as outlined above, you can use weak import for the `ThreadsLib` in PowerPC code. Weak import allows your application to launch even if the Thread Manager is unavailable. If your application requires threads, you can alert the user and quit gracefully.

Create the main thread

The main thread is a thread that represents the principal flow of control in your application. It should contain your main event loop. This is the thread that's executing when your application starts up. Even though the Thread Manager automatically creates a thread for every application, you must still create a PowerPlant thread object for the main thread of control in your application.

Typically you create the main thread in the application object's constructor. The easiest way to create a main thread object is to instantiate a UMainThread object. The code snippet below shows how.

```
MyApp::MyApp()
{
  ...
  LThread *myMainThread = new UMainThread;
}
```

If you need to customize the behavior of the main thread, you can derive a class from LThread or from UMainThread. For example, if you want to perform special actions when the main thread swaps in or out, override the `SwapContext()` member function.

WARNING! There are two considerations to keep in mind with respect to the main thread. First, you must create the main thread before any other threads. Second, do not call the main thread's `Run()` or `Resume()` functions. This thread is already running! You should not even implement a `Run()` function for the main thread.

Creating Threads

There are two principal issues involved with instantiating a thread object:

- Allocating memory for a thread—general allocation issues
- Calling the LThread constructor—details of the LThread constructor

Allocating memory for a thread

When creating thread objects, keep in mind that each thread actually requires two allocations: one for the thread object itself, the other for data maintained by the Thread Manager (mostly for the thread's stack).

Thread objects *must* be allocated dynamically (using `operator new`). It is illegal to create a static or automatic thread object, or to embed one within another object. Pointers and references to threads may, of course, be allocated automatically or included as data members in other objects.

If you intend to use threads from a preallocated pool, you must call `AllocateThreads()` to create the pool.

Calling the LThread constructor

You cannot instantiate an LThread object directly, because LThread is an abstract class. You must instantiate objects based on a class derived from LThread. This might be LSimpleThread, or a thread class of your own design.

In your class constructor, you initialize any data members and perform any other task unique to your derived class. This is the same as it is for any constructor. In addition, you must call the LThread constructor in the constructor's initializer list.

The parameters you pass to the LThread constructor determine if the thread is cooperative or preemptive (it should always be cooperative), the size of the thread's stack, how the Thread Manager allocates memory for the thread, and where the Thread Manager will store the result of the thread.

The LThread constructor looks like this:

```
LThread(Boolean inPreemptive,
        Uint32 inStacksize = thread_DefaultStack,
        LThread::EThreadOption inFlags =
                        threadOption_Default,
        void **outResult = NULL)
```

Table 3.8 lists the parameters, default values, and the purpose of each parameter.

Table 3.8 LThread constructor parameters

Parameter	Default value	Purpose
`inPreemptive`	none, you should always use false	`false` = cooperative
`inStacksize`	`thread_DefaultStack`	size of thread stack
`inFlags`	`threadOption_Default`	controls memory allocation
`outResult`	`NULL`	thread result

You should always set `inPreemptive` to false. The Thread Manager no longer supports preemptive threads.

With respect to stack size, you might want to use the default value early in development. The default stack size is usually sufficient. Later, you can monitor your stack usage and adjust the stack size accordingly. The 680x0 and PowerPC processors have different stack requirements.

PowerPlant defines several additive flags that you can combine in the `inFlags` parameter to control how the Thread Manager allocates memory for the thread you are creating. Table 3.9 lists the flags and their effects.

Table 3.9 Memory allocation flags for threads

`threadOption_Default`	The thread's stack is explicitly allocated by the Thread Manager and the FPU registers are preserved across context switches.
`threadOption_NoFPU`	Do not save and restore the FPU registers for this thread. Has no effect on the PowerPC.
`threadOption_UsePool`	Allocate the thread's stack from the Thread Manager's internal memory pool. You preallocate threads in this pool by calling `LThread::AllocateThreads()`.
`threadOption_Exact`	Forces the Thread Manager to allocate a stack with *exactly* the requested size.
`threadOption_Alloc`	If the Thread Manager has exhausted its internal memory pool, it returns an error. This flag causes the constructor to allocate the stack by calling the Memory Manager instead.
`threadOption_Main`	Create the application's main thread. When this flag is set, all of the other flags are ignored. Memory isn't actually allocated for the thread's stack, because the main thread is actually using the normal application stack instead.

Finally, the `outResult` parameter is the address of a 4-byte storage area where the Thread Manager will place the thread's result value. You may retrieve and change a thread's result with `GetResult()` and `SetResult()`.

Running a Thread

Except for the main thread, a thread is created in the suspended state. The thread will not start executing until it is put into the ready state. You do this by calling `Resume()` after creating the thread. For example:

```
MyThread * myThread = new MyThread(false,
      thread_DefaultStack, threadOption_Default);
myThread->Resume();
```

Do *not* call the thread's `Run()` function. The `Run()` function starts executing automatically the first time the thread becomes the current thread.

Although you never call it, all concrete thread classes must define a `Run()` function that has the following prototype:

```
virtual void *Run(void);
```

The Thread Manager calls this function when the thread starts execution. The contents of this function define the behavior of the thread. The `Run()` function may call any other application service. It does not have to be self-contained.

See "The Run() function" on page 55 for a discussion of what happens when the `Run()` function returns.

It is often useful to associate data with a thread. You can do this easily by deriving a class from LThread and adding data members to hold the necessary data. You can then access these members normally from within the `Run()` function.

Modifying Thread State

Once a thread is running, you use member functions to change the thread's state.

The simplest state change is to transfer control to another thread. Cooperative threads must call `Yield()` often in order to give other threads a chance to run.

You can also suspend a thread. A suspended thread receives no time until you resume it. For example, the following snippet suspends the main thread, does some processing, and then resumes the main thread. It isn't usually advisable to suspend the main thread, be-

cause the main thread typically manages the user interface and main event loop. This snippet is for illustrative purposes only.

```
void *MyThread::Run(void)
{
  LThread *mainThread = LThread::GetMainThread();
  // the main thread stops here
  mainThread->Suspend();

  // do some processing here

  mainThread->Resume();
  // the main thread continues here
  return (NULL);
}
```

You can also put a thread to sleep for a period of time. The following snippet executes some code after a four second delay.

```
void *MyThread::Run(void)
{
  while (true)
  {
    Sleep(4000); // go to sleep for 4 seconds
      // do some processing here
  }
    return (NULL);
}
```

You may also change a thread's state to waiting or blocked. See the Data Coherency and Asynchronous Operations topics in this chapter for information on using those thread states.

Deleting Threads

A thread is deleted automatically when the `Run()` function completes. See "The Run() function" on page 55.

If you want to kill a thread before it completes, call the thread's `DeleteThread()` member function. Do *not* call `operator delete` for a thread object. Deleting a thread in PowerPlant requires a lot of cleanup. The `LThread::DeleteThread()` function takes care of this cleanup for you.

Do not delete the main thread. Attempting to do so will cause an exception.

PowerPlant takes care of the details of deleting threads for you. In most cases, the thread is destroyed immediately and its memory released. However, what actually happens depends upon the thread's state, and also on the current thread. In certain cases, destruction or memory release may be delayed. There is a storage reclamation thread that handles the delayed release of memory. Table 3.10 lists what happens when you delete a thread in various states.

Table 3.10 Effect of deleting threads in various states

State	Effect
Current	Destroyed and deallocated immediately. The call to `DeleteThread()` never returns.
Ready	Destroyed and deallocated immediately.
Suspended	Destroyed and deallocated immediately.
Sleeping	Destroyed and deallocated immediately.
Waiting	Removed from its semaphore's queue of waiting threads. It is then destroyed and deallocated.
Blocked	Destroyed and deallocated the next time the thread is switched in (which means the thread will be current at that time). This will happen at some moment after its async call completes.

Data Coherency

With multiple threads comes the issue of data coherency. A threaded application must deal with the possibility that two threads may try to access, and possibly modify, shared data.

If a thread uses only data local to the thread, data coherency is not a problem. However, such threads are rare. To be effective, most threads must use other application services and data. It might be a global variable, a shared data structure, or some other form of data that exists and persists in a context outside of and shared by multiple threads.

There are several possible solutions to ensure that any shared data used by a thread is reliable. The four strategies supported in PowerPlant include:

- Criticality—locking out all other threads in critical operations
- Context Switching—preserving shared data internally
- Using Semaphores—flagging data as in use
- Inter-Thread Communication—passing data from one thread to another

Criticality

You can prevent other threads from taking command by simply refusing to yield. However, relying on the absence of a yield is unwise. If you call some function that ultimately results in an unanticipated yield, your code could crash. It is not always easy to foresee every possible path of execution, so it can be difficult to rule out all chance of a yield occurring.

A wiser way to protect data is to lock out all other threads during critical operations. Declaring a critical operation is the simplest mechanism to ensure that data does not change while a thread uses it. If your thread is about to access shared data, you can seize control of the process and prevent any thread from switching into control.

Simply call the thread's `EnterCritical()` function at the beginning of the operation. At the end of the operation, call `ExitCriti-`

cal(). Between these two calls, you are guaranteed that no other thread will gain control. These calls can be nested.

Alternatively, you can declare an StCritical object at the beginning of the critical block. The constructor calls the Thread Manager's ThreadBeginCritical(). The destructor calls ThreadEndCritical().This has an added advantage in that the StCritical object destructor is called even if an exception is thrown within the critical code.

The limitation of this approach is that you should not perform time-consuming operations in a critical block. Locking out all other threads defeats the purpose of a threaded strategy. However, these calls are very useful if you are going to use shared data for a very brief time.

Context Switching

Each thread has a context—the state information associated with that thread. When a thread becomes current, the Thread Manager switches in the thread's context. The previous thread's context is switched out.

If your thread uses a global variable, you can preserve a local copy of the value when the thread goes out of context, and restore it when the thread goes back into context. You can override the SwapContext() function to do this. PowerPlant calls this function whenever a thread is in the process of being switched in or out.

This process is analogous to preserving and restoring the A5 world during certain process switches. Listing 3.1 illustrates how to adjust when swapping thread context.

Listing 3.1 Context switching code

```
// global that must preserved for each thread
extern Boolean gAnImportantGlobal;

class MyThread : public LThread
{
```

```
  public:
  // member variables that preserve the global
  Boolean mSavedGlobal, mTemp;
  ...
                MyThread();
  virtual void  SwapContext(Boolean swappingIn);
  ...
};

MyThread::MyThread() : LThread(false)
{
  // initialize our member variable
  mSavedGlobal = gAnImportantGlobal;
}

void MyThread::SwapContext(Boolean swappingIn)
{
  if (swappingIn) // thread is being switched in
  {
    // first, call inherited swap function
    LThread::SwapContext(swappingIn);

    // then, do custom swap-in action;
    // we stuff the global with our saved value
    mTemp = gAnImportantGlobal;
    gAnImportantGlobal = mSavedGlobal;
  }
  else // the thread is being switched out
  {
    // first, do custom swap-out action;
    mSavedGlobal = gAnImportantGlobal;
    gAnImportantGlobal = mTemp;

    // then, call inherited swap function
    LThread::SwapContext(swappingIn);
  }
}
```

If you examined this sample code, you noticed that it called the inherited `SwapContext()` function. The inherited function handles

vital values related to the A5 world, exception handling, the `sCurrentThread` data member, and performs other critical work.

WARNING! If you override `SwapContext()`, you must call the inherited `SwapContext()` function if you expect a threaded application to function properly.

Using Semaphores

Semaphores are really quite simple once you understand them.

A general-purpose *semaphore* (or flag) is simply a counter associated with data in an object. In PowerPlant, if the counter is greater than zero, the data is available. If the counter is zero or less, the data is not available. It's that simple.

If you want to attach a semaphore to data, you typically create a semaphore data member in the object. You may also create an independent semaphore object and associate it with the data you want flagged. The initial count provided to the semaphore determines the number of threads that can simultaneously access the semaphore.

Just before you access the shared data, you call the semaphore object's `Wait()` function. If the data is available (the counter is greater than zero), this call decreases the counter by 1 and returns immediately. This gives you access to the data.

If the counter is already zero or negative when you call `Wait()`, the data is unavailable. The calling thread is entered into the semaphore's list and put into the waiting state.

A thread can specify the amount of time it is willing to wait when it calls the `Wait()` function. If the time expires before the thread gets access, the wait times out and an error is returned.

After you are through accessing the data, you call the semaphore object's `Signal()` function. This increases the counter by 1. If the count becomes positive and there are threads waiting on the semaphore, one of the waiting threads is returned to the ready state. A

subsequent context switch will give control to that thread. At that time, the thread that had been waiting (but is now current) will return from the call to `Wait()` and have access to the data.

In a nutshell, a semaphore is an automatic method of putting a thread in the wait state until the flagged data is accessible.

PowerPlant provides three semaphore classes. LSemaphore implements a general-purpose semaphore as described above. LEventSemaphore is useful when two or more threads need to be synchronized. Its distinguishing property is that when the semaphore is made available, *all* waiting threads are released simultaneously. LMutexSemaphore implements a *mutual exclusion*, or mutex, semaphore. This type of semaphore allows only one thread to claim the semaphore at any one time. It is very useful for implementing shared data structures.

Listing 3.2 illustrates how to use LMutexSemaphore on a simple implementation of a shared stack. In this example the stack contains LLink objects, but it could contain data of any type.

Listing 3.2 Mutually exclusive access to shared data

```
class CSimpleSharedStack : public CSimpleStack {
public:
                CSimpleSharedStack();
  virtual void  Push(LLink *data);
  virtual LLink* Pop(void);

private:
  LMutexSemaphore fAccess; // access to stack
};

CSimpleSharedStack::CSimpleSharedStack() : fAccess(FALSE) {}

void CSimpleSharedStack::Push(LLink *data)
{
  // wait for access
  fAccess.Wait();
  // do the work
  CSimpleStack::Push(data);
```

```
  // we've finished
  fAccess.Signal();
}

LLink *CSimpleSharedStack::Pop(void)
{
  LLink *data;

  fAccess.Wait();
  data = CSimpleStack::Pop();
  fAccess.Signal();
  return (data);
}
```

Calls to `Wait()` and `Signal()` bracket the calls to the base class that do the actual work. It is guaranteed that no two threads can ever simultaneously execute the code in between these two calls.

Notice that both the `Push()` and `Pop()` routines use the same semaphore. Because the semaphore applies to the entire stack, if one thread is pushing data onto the stack, no other thread can be pushing or pulling data off the stack. Hence the term mutual exclusion.

Imagine this semaphore code is not in place. Now suppose a thread calls the `Pop()` function. While executing the `Pop()` function, the thread is preempted by a second thread that calls the `Push()` function. Unfortunately, data has not been fully removed by the thread calling the `Pop()` function, and the stack pointer hasn't been updated yet. As a result, the thread calling the `Push()` function damages the stack. Such a disastrous occurrence is perfectly possible in threaded code.

Mutual exclusion is such a useful strategy that PowerPlant includes the StMutex utility class to make writing the code easier, and to release the semaphore even if the code throws an exception. Simply declare an StMutex object in the block where you want the data protected. The StMutex constructor waits on a semaphore and the destructor signals it. Using this class, the `Push()` function would be written as follows:

Listing 3.3 Using StMutex

```
void CSimpleSharedStack::Push(LLink *data)
{
  // mutex is constructed by waiting on fAccess
  StMutex mutex(fAccess);
  CSimpleStack::Push(data);

  // before returning, mutex is destroyed
  // by signalling fAccess
}
```

Semaphores can also be used to signal other threads that data is not ready. For example, assume you are using the stack to pass data from one thread to another. The stack remains empty until the data is prepared and pushed onto the stack. Assume that if an attempt is made to pop data from the empty stack, the stack returns `NULL`.

A thread needing information from the stack in order to proceed would probably end up with code like this:

```
  // wait for some data from another thread
  while ((myLink = myStack->Pop()) == NULL)
    LThread::Yield();
  // got the data -- now process it
```

Even if the thread yields control to other threads from within its loop, it is still polling the stack object repeatedly. In fact it is *busy-waiting*. In the world of concurrent programming, this is a Very Bad Thing. It uses up CPU time without performing any useful work.

You can use an availability semaphore to create a more satisfactory solution. Set the semaphore whenever the stack is empty. Then, rather than spinning its wheels in a busy-waiting state and wasting processor time, the thread is put into a wait state until the data is ready. Listing 3.4 shows how to do this.

Listing 3.4 Dealing with an empty stack

```
class CSafeSharedStack:public CSimpleSharedStack {
public:
                 CSafeSharedStack();
  virtual void   Push(LLink *data);
  virtual LLink* Pop(void);

private:
  LSemaphore fValueAvailable; // is stack empty
};

CSafeSharedStack::CSafeSharedStack() : fValueAvailable(0) {}

void CSafeSharedStack::Push(LLink *data)
{
  CSimpleSharedStack::Push(data);
  fValueAvailable.Signal();
}

LLink *CSafeSharedStack::Pop(void)
{
  fValueAvailable.Wait();
  return (CSimpleSharedStack::Pop());
}
```

The `fValueAvailable.Signal()` function is called every time something is pushed onto the stack. Likewise, `fValueAvailable.Wait()` is called every time something is popped from the stack. Hence, `fValueAvailable`'s integer count encodes the number of elements on the stack. If the count is non-positive, calls to `Pop()` will wait until a subsequent call to `Signal()` clears the semaphore and puts the waiting thread back in the ready state. The busy-waiting loop shown above is replaced by a call to `Pop()`.

Inter-Thread Communication

In the Using Semaphores section the example code demonstrated how to use a mutual exclusion semaphore to pass data from one thread to another. LSharedQueue does this for you automatically.

Suppose that an application contains two threads that need to share data. Because the threads execute asynchronously in relation to one another, passing data from one to the other becomes tricky. A straightforward solution is to decouple the data from both threads and place it in a thread-safe shared data structure.

Use an LSharedQueue object as a shared channel of communication. Because LSharedQueue stores LLink objects, you can create a linked list of any kind of data you want in the queue. Your data objects descend from LLink and add whatever data members are necessary.

When one thread has data for the other, it puts the data in the queue by calling `LSharedQueue::NextPut()`. The other thread retrieves the data by calling `LSharedQueue::Next()`. LSharedQueue uses semaphores to ensure that the queue is thread safe. The strategy is very much like that outlined in the Using Semaphores section.

Listing 3.5 demonstrates a typical pattern of data sharing, the producer-consumer relationship. One thread produces data that is consumed by the second thread.

Listing 3.5 Sample producer-consumer code

```
class MyProducerThread : public LThread {
public:
  MyProducerThread(LSharedQueue *inQueue) :
    LThread(false)
    { mQueue = inQueue; }

protected:
  virtual void Run(void);
  LSharedQueue* mQueue;
};

class MyConsumerThread : public LThread {
public:
  MyConsumerThread (LSharedQueue *inQueue)
              : LThread(false)
    { mQueue = inQueue; }

protected:
```

```
  virtual void Run(void);
  LSharedQueue *mQueue;
};

void MyMakeThreads(void)
{
  LSharedQueue *queue;
  MyProducerThread *producer;
  MyConsumerThread *consumer;

  // this queue is used as a communication
  // channel between the two threads
  queue = new LSharedQueue;

  // create the threads, pass in the queue as the
  // thread argument
  producer = new MyProducerThread(queue);
  consumer = new MyConsumerThread(queue);

  // fire 'em up
  producer->Resume();
  consumer->Resume();
}

void MyProducerThread::Run(void)
{
  LLink *data;

  while (TRUE)
  {
    // get some data
    data = MyGetData();

    // send it to consuming thread
    mQueue->NextPut(data);
  }
}

void MyConsumerThread::Run(void)
{
```

```
  LLink *data;

  while (TRUE)
  {
    // get data from producing thread
    data = mQueue->Next();

    // process it
    MyProcessData(data);
  }
}
```

In summary, you derive your data class from LLink and add your data to it. You can then put instances of your data class into the queue using `LSharedQueue::NextPut()` and retrieve them using `LSharedQueue::Next()`.

Asynchronous Operations

Setting up and running asynchronous operations is a process that goes hand-in-hand with threads. Threads attempt to provide for concurrent processes cooperatively. Asynchronous operations run at interrupt time, and allow you to implement the same kind of responsiveness for which threads are intended. You can start computationally intensive tasks running asynchronously, and regain control virtually immediately while the operation proceeds "in the background."

A problem may arise if a thread that begins an asynchronous operation must remain in existence until the operation completes. How does the thread know when the asynchronous operation is finished?

The LThread class contains functions that facilitate the use of asynchronous I/O. These functions obviate the need to write *any* code that runs at interrupt level. They also permit chained asynchronous system calls in a way that simplifies program maintenance.

NOTE: Read “Asynchronous Routines on the Macintosh” in issue 13 of develop before attempting thread-blocking I/O.

For example, a thread’s `Run()` method that reads data from a first file and writes it to a second file might be written like this:

Listing 3.6 Asynchronous read and write

```
class MyFileCopyThread : public LThread
{
  Int16 mInRefNum, mOutRefNum;
  ...
  virtual void *Run(void);
};

void *MyFileCopyThread::Run(void)
{
  SThreadParamBlk  pb;
  char             buff[512];
  OSErr            err;

  // Fill parameter block
  pb.ioPB.F.ioParam.ioRefNum = mInRefNum;
  pb.ioPB.F.ioParam.ioBuffer = buff;
  pb.ioPB.F.ioParam.ioReqCount = sizeof(buff);
  pb.ioPB.F.ioParam.ioPosMode = fsFromStart;
  pb.ioPB.F.ioParam.ioPosOffset = 0;

  // Install special I/O completion routine that
  // resumes the thread. Needs to be done once.
  SetupAsynchronousResume(&pb);

  // start async read
  // note that we ignore the return code
  (void) ::PBReadAsync(&pb.ioPB.F);

  // If there is no error, block the thread. Upon
  // completion of the call, the thread will be
  // resumed and returns the error code from the
```

```
  // I/O parameter block.
  err = SuspendUntilAsyncResume(&pb, noErr);

  if (err != noErr)
  {
    // Handle read errors
  }

  pb.ioPB.F.ioParam.ioRefNum = mOutRefNum;
  pb.ioPB.F.ioParam.ioReqCount =
          pb.ioPB.F.ioParam.ioActCount;

  // Start async write & block thread
  (void) ::PBWriteAsync(&pb.ioPB.F);
  err = SuspendUntilAsyncResume(&pb, noErr);

  if (err != noErr)
  {
    // Handle write errors
  }

  return (NULL);
}
```

Note that the I/O parameter block, which is embedded in the `SThreadParamBlk` structure, is allocated on the stack (as is the I/O buffer).

Why are we ignoring the result from `PBReadAsync` and `PBWriteAsync`? Because async File Manager calls return garbage. For more information, consult the develop article mentioned at the beginning of this topic.

If you are calling a manager that returns meaningful result codes, you should pass the result code to `SuspendUntilAsyncResume()`. For example:

```
SThreadParamBlk pb;
SetupAsynchronousResume(&pb);
```

```
error = ::PRegisterName(&pb.ioPB.M);
SuspendUntilAsyncResume(&pb, error);
```

What happens if the async call completes before returning to its caller? Because the thread is resumed from within the async call's completion routine, one might expect that the subsequent call to `SuspendUntilAsyncResume()` would leave the thread waiting for a resume that had already occurred—meaning that the thread would never wake up.

Luckily, the async call's completion routine is a little smarter than that. When it detects that the thread isn't yet suspended, it sets up a Time Manager task that delays for a short period of time. Between the time the completion routine returns and the time the Time Manager task fires, the calling thread will perhaps have had time to suspend itself. If so, the Time Manager task will resume it. If not, the cycle will begin again.

The `SThreadParamBlk` structure contains a union of the more commonly used I/O parameter blocks. If you want to use a parameter block that isn't supported in `SThreadParamBlk`, you can declare your own. For example, if you want to make asynchronous calls to the PPC Toolbox, you can declare a structure like this:

```
typedef struct {
  LThread *ioThread;
  Int32 ioGlobals;

// important: the param block must be preceeded
// by the two io variables!

  PPCParamBlockRec ioPB;
} MyPPCParamBlk;
```

When you perform asynchronous I/O, you just typecast your structure to a `SThreadParamBlk`:

```
MyPPCParamBlk pb;

SetupAsynchronousResume(
```

```
  (SThreadParamBlk *) &pb, myPPCUPP);
  (void) ::PPCInformAsync(&pb.ioPB.informParam);
    err = SuspendUntilAsyncResume(
                (SThreadParamBlk *) &pb, noErr)
```

Note that a UPP was supplied to `SetupAsynchronousResume()`. Why? Because PPC Toolbox completion routines don't use the same calling convention as, say, File Manager completion routines. All of the I/O calls supported by the members of `SThreadParamBlk` use the "pointer to parameter block in A0" calling convention. If you make any async call whose completion routine uses different calling conventions, you need to supply a UPP. This UPP will be used as the completion routine of the asynchronous call. It is your responsibility to allocate and properly initialize this UPP. In the example above, the actual procedure pointed to by the UPP might look like this:

Listing 3.7 A sample completion proc

```
#include <stddef.h>

pascal void MyPPCToolboxCompletionProc
                          (PPCParamBlockRec *ppcPB)
{
  MyPPCParamBlk *myPB;

  myPB = (MyPPCParamBlk *)
      (-offsetof(MyPPCParamBlk, ioPB)
      + (char *) ppcPB);

  // note that the PPC Toolbox has set up
  // the A5 world
  LThread::ThreadAsynchronousResume(
                                    myPB->ioThread);
}
```

In cases where asynchronous routines do not use the standard I/O parameter block, you cannot call `LThread::SetupAsynchronousResume()` nor `LThread::SuspendUntilAsyncResume()`

but must instead block the thread yourself. For example, the following code sets up a speech channel, initiates speech synthesis, then suspends itself until the text has been completely spoken (see the Speech Manager documentation for more information).

Listing 3.8 Example of asynchronous use of the Speech Manager

```
void MyMakeSpeech(VoiceSpec *voice, Ptr text,
                  long textLen)
{
  extern pascal void EndSpeechProc(SpeechChannel,
                  long refCon);

  LThread *thread = LThread::GetCurrentThread();
  SpeechDoneUPP speechDoneUPP;
  SpeechChannel *chan;
  OSErr err;

  // allocate UPP for callback
  speechDoneUPP = NewSpeechDoneProc(EndSpeechProc);

  // allocate speech synthesis channel
  err = NewSpeechChannel(voice, &chan);

  // set up callback parameters
  err = SetSpeechInfo(chan, soCurrentA5,
                    (void *) SetCurrentA5());
  err = SetSpeechInfo(chan, soRefCon, thread);
  err = SetSpeechInfo(chan, soSpeechDoneCallBack,
                    speechDoneUPP);
  // talk
  err = SpeakText(chan, text, textLen);

// Suspend ourselves until we're done speaking.
// Note that we use the Block call instead of the
// Suspend call. This will prevent the thread from
// being killed before the async call completes.

  if (err == noErr)
    thread->Block();
```

```
  // we've been resumed via the callback
  // (or SpeakText returned an error)
  err = DisposeSpeechChannel(chan);

  // clean up UPP
  DisposeRoutineDescriptor(speechDoneUPP);
}

// This function called at interrupt time by the
// Speech Manager when all of the text has been
// spoken.

pascal void EndSpeechProc(SpeechChannel, long refCon)
{
  // note that the Speech Mgr has set up our A5 world

  LThread::ThreadAsynchronousResume(
                                (LThread *)refCon);
}
```

Note that `LThread::ThreadAsynchronousResume()` is the *only* function in the PowerPlant threads classes that may be called from interrupt-level code. This means that it is illegal to access any other member function or variable of an object belonging to the threads classes from within an I/O completion or other interrupt-level routine.

Summary of Threads in PowerPlant

The field of concurrent programming is full of possibilities and complications. PowerPlant provides classes that help you master the problems and realize the benefits of a threaded strategy.

You can create simple threads easily using LSimpleThread, and UMainThread. You can extend these threads easily by deriving from LThread or LSimpleThread and adding your own data.

If your threads share data, PowerPlant provides you with a set of semaphore classes that easily and almost automatically protect shared data from inadvertent disaster.

If your threads must communicate shared data, the LLink, LQueue, and LSharedQueue provide a simple mechanism. In fact, the LLink and LQueue classes can be used as a general purpose FIFO queue for any kind of data.

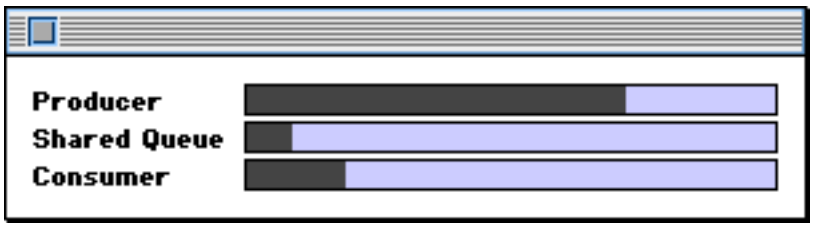
The code exercise for this chapter uses LSharedQueue to facilitate communication between two threads.

Code Exercise for Threads

In this exercise you create an application that uses threads. Each thread has a visual representation, so you can see how far along it is in its task.

The PPob resource and visual interface for this application have been provided for you. Each window you create has three progress bars, as shown in Figure 3.3. One thread produces data, the other consumes it. The threads communicate with each other via a shared queue. As the producer thread does its work, the progress bar empties. The queue begins to fill, and then the consumer thread starts taking data off the queue.

Figure 3.3 Threads



The progress bar code is provided for you. Each of these objects—the two threads and the queue—has its own progress bar object.

In this exercise you write the code to initialize threads in PowerPlant, instantiate the necessary threads, start the threads running, and destroy the threads. You also create the shared queue.

Before starting, examine the `main()` function in `CThreadsApp.cp` to see how the code checks for the Thread Manager at startup and exits gracefully. In addition, look at how the ThreadsLib file is imported in the project file. It uses weak import so that the application can launch even in the absence of the Thread Manager.

1. Prepare for threads.

`CThreadsApp()` `CThreadsApp.cp`

To accomplish this task, you create a main thread and attach an LYieldAttachment to the application object. A handy place to accomplish both tasks is in the application object constructor.

```
// Create the main thread.
new UMainThread;

// Add a yield attachment.
AddAttachment( new LYieldAttachment( -1 ) );
```

Note that the attachment is set to -1, so it yields immediately when called. This attachment gets control from the main event loop. The main event loop is part of the main thread. The net effect of the attachment is to ensure that the main thread yields regularly—once for every event received.

2. Create the threads and queue.

`FinishCreateSelf()` `CThreadWindow.cp`

In this step you first create a queue object, and then the two thread objects. For each object you get its associated progress pane, and then instantiate the object. After that, you start each thread running.

The CThreadWindow class has three data members, `mSharedQueue`, `mProducerThread`, and `mConsumerThread`. Feel free to explore the class declaration.

a. Create the shared queue.

The queue constructor requires a pointer to a CProgressPane object. The existing code gets that pointer for you. After that, allo-

cate a new CVisualSharedQueue object. Store the pointer to the new queue in the `mSharedQueue` data member.

b. Create the producer thread.

The existing code gets a pointer to the progress pane for this object. The constructor requires both the shared queue and the progress pane. Allocate a new CProducerThread object. Store the pointer to the new thread object in the `mProducerThread` data member.

c. Create the consumer thread.

The consumer thread is an object of the CConsumerThread class. The existing code gets a pointer to the progress pane for this object. The constructor requires both the shared queue and the progress pane. Allocate a new CConsumerThread object. Store the pointer to the new thread object in the `mConsumerThread` data member.

d. Make each thread ready.

Call each thread's `Resume()` function. The code for all four substeps is listed below.

```
// Create the shared queue.
mSharedQueue = new CVisualSharedQueue(
                                  theProgressPane );
ThrowIfNil_( mSharedQueue );

// Get the producer's progress pane.
theProgressPane = (CProgressPane *)
FindPaneByID( kProducerProgressPane );
Assert_( theProgressPane != nil );

// Create the producer thread.
mProducerThread = new CProducerThread(
                 mSharedQueue, theProgressPane );
ThrowIfNil_( mProducerThread );

// Get the consumer's progress pane.
theProgressPane = (CProgressPane *)
FindPaneByID( kConsumerProgressPane );
Assert_( theProgressPane != nil );
```

```
// Create the consumer thread.
mConsumerThread = new CConsumerThread(
                  mSharedQueue, theProgressPane );
ThrowIfNil_( mConsumerThread );

// Start the threads.
mProducerThread->Resume();
mConsumerThread->Resume();
```

3. Write a thread constructor.

`CProducerThread()` `CProducerThread.cp`

In this step you write a CProducerThread constructor. The CConsumerThread constructor is identical. It has been provided for you.

The CProducerThread constructor must call the LThread constructor. Set the LThread parameters properly. Make the thread a cooperative thread. Don't forget, several of the LThread constructor parameters have default values that you may find acceptable. Then initialize the CProducerThread data members. The value for the two data members are passed in as parameters to the call.

```
CProducerThread::CProducerThread(
  LSharedQueue* inQueue,
  CProgressPane* inProgressPane )
    : LThread( false ), mQueue( inQueue ),
      mProgressPane( inProgressPane )
{
}
```

This code creates a cooperative thread, and uses default values for all other LThread constructor parameters. It initializes the values of `mQueue` and `mProgressPane` appropriately.

4. Write the `Run()` function for the producer.

`Run()` `CProducerThread.cp`

Because this is a demonstration, a couple of unusual things happen with the `Run()` function for the producer thread.

First, this thread simply creates an LLink object and puts it on the queue. There is no real data attached to the LLink object.

Second, the state of the progress bar actually controls the thread, rather than the other way around. The producer progress bar starts out full and becomes empty over time. When the progress bar is empty, the thread suspends itself. The function repeatedly puts data on the queue and decrements the progress bar.

The `Run()` function should do four things.

a. Suspend the thread upon completion.

When `Run()` returns, the thread deletes itself. In this case you don't want the thread to delete itself. You'll delete the thread when you close the window that contains the thread, because this thread is attached to a visual object.

The existing code has an if test for `theValue`. If `theValue` is less than or equal to the minimum value, call `Suspend()` to suspend the thread.

b. Put data in the shared queue.

Use the `mQueue` object's `NextPut()` function and pass it a new LLink object.

c. Decrement the progress bar value.

Send the `mProgressPane` a `SetValue()` message. The local variable `theValue` holds the current value. Decrement `theValue` by one before passing it.

d. Yield to other threads.

Simply call `Yield()`.

```
if ( theValue <= mProgressPane->GetMinValue() )
{
  Suspend();
}

// Put data in the shared queue.
```

```
mQueue->NextPut( new LLink );

// Decrement the progress bar value.
mProgressPane->SetValue( theValue - 1 );

// Yield so that other threads may get time.
Yield();
```

5. Write the `Run()` function for the consumer thread.

`Run()` `CConsumerThread.cp`

In this step you write the consumer thread's `Run()` function. This is the converse of the code you wrote in the previous step.

Because this is a demonstration, you simply delete the data you retrieve. There are five tasks to perform.

a. Suspend the thread upon completion.

When `Run()` returns, the thread deletes itself. In this case you don't want the thread to delete itself. You'll delete the thread when you close the window that contains the thread, because this thread is attached to a visual object.

The existing code has an if test for `theValue`. If `theValue` is greater than or equal to the minimum value, call `Suspend()` to suspend the thread.

b. Get data from the shared queue.

Send the `mQueue` object a `Next()` message to get the next link. Receive the return value in a local `LLink*` variable.

This is really the meat of of the entire process. The call to `Next()` will suspend the thread until data is available.

c. Delete the data.

Delete the LLink object you just retrieved. This line of code won't execute until after the call to `Next()` returns (which means the data was available, the thread has become active, and the data has been retrieved.

d. Increment the progress bar value.

Send the `mProgressPane` a `SetValue()` message. The local variable `theValue` holds the current value. Increment `theValue` by one before passing it.

e. Put this thread to sleep.

Rather than yield, put this thread to sleep for a brief period. This slows the thread down with respect to the producer, so you can see data accumulate in the shared queue. Remember that putting the thread to sleep also causes a yield, so you accomplish two things at once. Fifty milliseconds is a good time to sleep.

```
if ( theValue >= mProgressPane->GetMaxValue() )
{
  Suspend();
}

// Get data from the shared queue.
LLink* theData = mQueue->Next();

// Just delete it.
delete theData;

// Increment the progress bar value.
mProgressPane->SetValue( theValue + 1 );

// Go to sleep for a while.
Sleep( 50 );
```

6. Destroy the threads.

`~CThreadWindow` `CThreadWindow.cp`

The threads never return because you suspend them before they return. Therefore, they do not delete themselves. You destroy these threads when you close the window that contains them.

Simply call `DeleteThread()` for both threads in the window. The existing code then cleans up the shared queue.

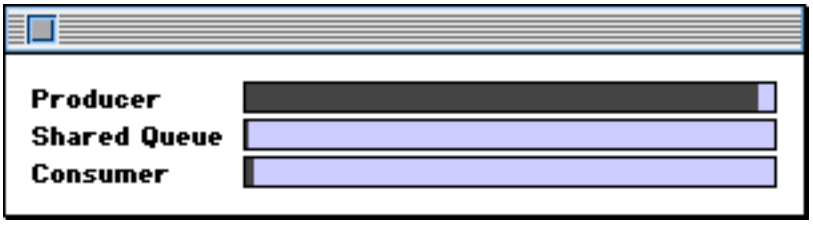
```
// Delete the producer thread.
if ( mProducerThread != nil )
    mProducerThread->DeleteThread();
```

```
// Delete the consumer thread.
if ( mConsumerThread != nil )
    mConsumerThread->DeleteThread();
```

7. Build and run the application

When the project builds correctly and you run the application, a window appears as shown in Figure 3.4. The producer thread progress bar starts out full. It immediately begins to empty as the thread it represents places data on the queue. Shortly after that, the consumer thread starts to remove data from the queue.

Figure 3.4 Threads in operation



Next, make several windows. In the process two important things are happening, one obvious, one subtle. The obvious feature of threads is that no matter how many windows you make, all the threads operate concurrently. The Thread Manager and PowerPlant work together to ensure that all threads get time. This is cool.

The subtle feature of threads is the responsiveness of the application. After you make one window, the application is busy calculating and retrieving data. In a non-threaded application, once that process begins, the application is tied up and unresponsive. In a threaded application, the user interface (the main event loop) is also part of a thread. Therefore, even while computationally intensive operations are in progress, your application can still receive and handle events! This is *very* cool.

Finally, remember that the shared queue implements a semaphore that protects the integrity of the queue. Experiment with various sleep times for both the producer and consumer thread, and put this mechanism to the test. As it is now, the producer creates threads

faster than the consumer retrieves them. Slow down the producer and speed up the consumer. The consumer will be put into a waiting state automatically until data appears. The queue may always appear empty, because the consumer will remove data as soon as it appears on the queue. However, the consumer does not become "busy-waiting" and the data queue remains safe and intact.

If you'd like to explore further, here's a problem you can solve. While the threads are running in one or more windows, open the application's About box. What happens to the threads? They stop dead in their tracks. See if you can put the About box into a thread so that it doesn't seize control of the processor.

Here are two suggestions for how you might accomplish this task. A PowerPlant solution would be to use a PowerPlant-based movable modal dialog for the About box. Effectively, this makes the About box part of the application's main thread.

A non-PowerPlant solution would be to create an event filter proc for the modal dialog. This event filter would call `LThread::Yield()`.

Have a good time exploring.

4

Actions in PowerPlant

This chapter discusses how to implement undo and redo functionality using PowerPlant action classes.

Introduction to Actions in PowerPlant

From a user's perspective, the ability to undo an action is a wonderful feature in an application. We all make mistakes, and it is very nice when we can wipe out the mistake with a single keystroke or menu command. In addition, the ability to easily undo and redo an action lets the user toggle quickly between two alternate states. The user can then compare the two options and decide which is best.

In this chapter we discuss how you can implement undo functionality in a PowerPlant application. The ability to undo implies the ability to redo. In PowerPlant the two go hand-in-hand.

Implementing undo functionality in PowerPlant is remarkably straightforward, once you understand the strategy behind the PowerPlant approach. The topics discussed include:

- The Undo Strategy—the PowerPlant approach to undo
- Action Classes—the individual PowerPlant classes involved in undo
- Implementing Undo in PowerPlant—working with PowerPlant's action classes

The Undo Strategy

PowerPlant's undo strategy is based on the concept of an action. An action is a command or option that causes the state of the application to change. For example, if you issue a command to sort a list, you have changed the state of the data in the application.

PowerPlant encapsulates the concept of an action in the LAction class. An LAction object has both undo and redo capability, and preserves whatever data is necessary to restore the state of the application to its previous condition. For example, if you change a selected font, an LAction object might preserve the previous and the new font numbers or names. The object can then toggle between then two fonts as the user issues undo or redo commands.

As you know, commander objects handle commands in a PowerPlant application. When the user issues a command, the commander posts an action instead of acting directly. The process of posting the action also causes the action to occur for the first time. The action can then be undone and redone repeatedly.

The bridge between the LCommander object and the LAction object is an LUndoer object. The undoer is an attachment hosted by a commander. The undoer owns the action object. In response to the commander's messages, the undoer attachment tells the action to do, undo, or redo.

A commander typically has one undoer attachment. An undoer attachment may have one and only one action.

For more information on commanders, see Chapter 10 of *The PowerPlant Book*., "Commanders and Menus."

For more information on attachments, see Chapter 15 of *The PowerPlant Book*, "Periodicals and Attachments."

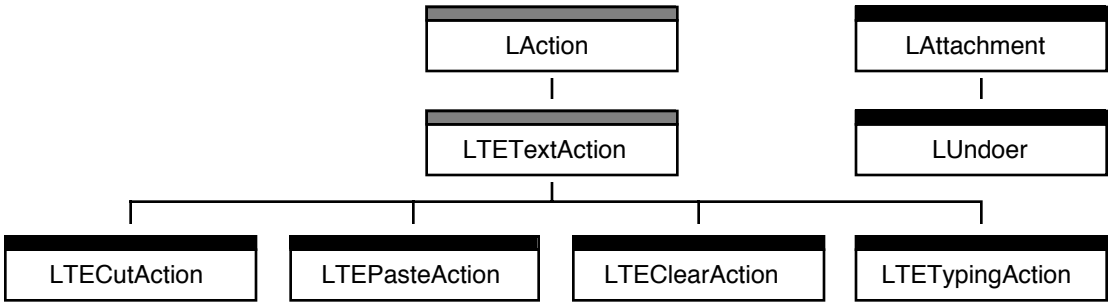
Action Classes

There are four major classes involved in PowerPlant's implementation of undo functionality. They are:

- LCommander—posts an action
- LAction—preserves state and implements do, undo, redo
- LUndoer—provides bridge between commander and action
- LTETextAction—implements actions for TextEdit operations

In addition, LTETextAction has four subclasses for cut, copy, paste, and typing actions, as shown in Figure 4.1.

Figure 4.1 LAction class hierarchy



Both LAction and LTETextAction are abstract classes.

The LUndoer class inherits from LAttachment. There are no classes in PowerPlant derived from LUndoer.

LCommander

The LCommander class is fully discussed in Chapter 10 of *The PowerPlant Book*. With respect to actions and implementing undo, there are two functions of the LCommander class of interest. They are:

- `PostAnAction()`
- `PostAction()`

`PostAnAction()` is a static member function that posts an action to the current target. Because it is a static function, it is available to any object that needs to post an action. If there is no current target, the function attempts to execute the action and then deletes the action. No undo is possible in this circumstance, because undo functionality requires an LUndoer object attached to a commander.

`PostAction()` is the function a commander uses to post an action. This function sends a `msg_PostAction` message to attachments. This gives the LUndoer an opportunity to respond. If no attachment intercepts this message (for example, because there is no attached undoer), then the function attempts to execute the action and then deletes the action. Again, no undo is possible in this circumstance, because undo functionality requires an LUndoer object attached to a commander.

LAction

LAction is an abstract base class that encapsulates do, undo, and redo behavior together with the data required to restore state.

LAction has three data members, listed in Table 4.1

Table 4.1 LAction data members

Data member	Stores
`mStringResID`	resource ID of `'STR#'` resource containing text for redo menu item(s)
`mStringIndex`	index number in the `'STR#'` resource for redo item related to this action
`mIsDone`	a boolean value to determine if the action is currently done or undone

When you implement undo, you typically modify the text of the **Undo** item in the Edit menu to reflect the nature of the command being undone or redone. For example, if your most recent action was to change the size of an object, the **Undo** item might read Undo Size. If you choose the **Undo** command, the text for the item would change to Redo Size.

When you create an LAction object, you specify the resource ID for the redo item text. The undo item text must be in a resource with an ID one greater than `mStringResID`. If your redo strings are in an `'STR#'` resource with ID 1000, your undo strings must be in an `'STR#'` resource with the ID 1001.

There are no data members used for storing application state. The information you preserve depends upon the nature of the action and the structure of your application. You would typically provide additional data members in subclasses derived from LAction.

Table 4.2 LAction member functions

Function	Purpose
`Redo()`	calls `RedoSelf()`, also used for doing action first time
`Undo()`	calls `UndoSelf()`
`CanRedo()`	returns `true` if action is currently undone
`CanUndo()`	returns `true` if action is currently done
`IsDone()`	returns value of `mIsDone`
`GetDescription()`	returns strings for undo and redo items
`IsPostable()`	returns `true`
`Finalize()`	called before deleting action, empty
`RedoSelf()`	pure virtual function
`UndoSelf()`	pure virtual function

The `IsPostable()` function always returns true in LAction. You should override this function to return false if an action cannot be undone (that is, the action is not undoable).

If you post a new action to an undoer, the previous action is deleted. Before deleting the action, PowerPlant calls `Finalize()`. This gives you an opportunity to do something with the action before it disappears. For example, you might want to preserve it to implement a multilevel undo.

Concrete subclasses of LAction must override `RedoSelf()` and `UndoSelf()` to implement the correct behavior.

You do not typically modify or override other functions in LAction. In fact, you do not even call any of these functions directly in a typical application.

LUndoer

If LAction is the core of PowerPlant's implementation of undo, LUndoer is the mover and shaker. LUndoer does most of the work of control and dispatch required for undo support.

There is only one data member, `mAction`. This data member stores a pointer to the action object that encapsulates the user's most recent action. Because the undoer owns the action, it can send messages to the action object telling it to redo or undo itself.

In a typical implementation of single-level undo you should never have to call or modify the LUndoer member functions Table 4.3 lists the member functions.

Table 4.3 LUndoer member functions

Function	Purpose
`ExecuteSelf()`	responds to commander messages
`FindUndoStatus()`	sets menu item text
`PostAction()`	handles a new action
`ToggleAction()`	tells action to undo or redo

As with all attachments, the `ExecuteSelf()` function intercepts messages for the host. LUndoer is always attached to a commander, and responds to three messages. They are:

- `msg_CommandStatus`—calls `FindUndoStatus()`
- `msg_PostAction`—calls `PostAction()`
- `msg_Undo`—calls `ToggleAction()`

Notice that the LUndoer attachment identifies and handles the situation when the user issues either an **Undo** or **Redo** command from

the Edit menu. Both come to the attachment as `msg_Undo`. The undoer knows which operation to perform based on the action's done state. If the action has just been done, the undoer tells the action to undo. If the action has just been undone, the undoer tells the action to redo itself.

NOTE: `LUndoer::PostAction()` is not the same as `LCommander::PostAction()`. `LUndoer::PostAction()` replaces the contents of `mAction` after calling `LAction::Finalize()`, and ensures that the menu item is updated.

LTETextAction

The LTETextAction class provides a basis for four PowerPlant action classes to support undo for text operations. The four PowerPlant classes derived from LTETextAction are:

- LTECutAction
- LTEPasteAction
- LTEClearAction
- LTETypingAction

There is no LTECopyAction. A copy operation does nothing except put a copy of data on the clipboard. Restoring the clipboard to its previous condition would be problematical at best. The data on the clipboard could very well have been placed there by another application and may exist in types your application does not understand.

These action classes follow the pattern discussed for an action class in general. Each has data members to store both the new and original data. Each has an `UndoSelf()` and `RedoSelf()` function to undo or do the action.

You can use these classes to support undo in TextEdit-based panes such as LEditField and LTextEdit. See the *PowerPlant Reference* and the source code for details on these classes. You can also study these classes to see practical implementations of the LAction concept.

Implementing Undo in PowerPlant

Implementing single-level undo in a PowerPlant application requires laying careful groundwork. After setting up the pieces, making it all run properly is trivial. There are three steps in this process:

- Create Action Classes
- Attach an Undoer
- Post an Action

This section also discusses how to:

- Implement Multilevel Undo

Create Action Classes

As your first step towards supporting undo functionality, you declare and define a series of subclasses of LAction. Each subclass should represent an action (command) that occurs in your application, typically an undoable action. If the action cannot be undone, you must override the `IsPostable()` function to return `false`.

In each subclass of LAction, you add the data members necessary to restore the application to its condition before the action is implemented. Because the undoer object toggles between two states (undone and redone), you typically save both the current state and the former state of the application in your action object. For example, if you move an object, you store both its former position and its new position in data members of the action class.

After that, you write the code that defines the `UndoSelf()` and `RedoSelf()` functions. Precisely what these functions do depends upon the nature of the action and how you store your data. In a typical implementation, each calls some function that makes something happen, and provides the necessary data to that function. In most cases, this is the function that you would have called directly from the commander before implementing undo.

Continuing with our movement example, to redo a move action you would call your routine that relocates the object to its new position.

To undo the routine, you call the same routine to relocate the object to its former position.

Don't forget to set up at least two `'STR#'` resources for the redo menu item text and the undo menu item text.

Attach an Undoer

When the user issues a command, some commander intercepts the command. It might be the application object or a subcommander, it doesn't matter.

Any commander that handles an undoable command must have an undoer attached to it for undo to work correctly. Typically you create the undoer when you create the commander. You might do this right after you successfully instantiate the commander. The code to do this is very simple:

```
theCommander->AddAttachment( new LUndoer );
```

This code snippet assumes that `theCommander` contains a pointer to an LCommander object of some type.

Post an Action

When the user issues a command (for example, by choosing a menu item), a commander's `ObeyCommand()` function gets control. In response to the command, you do two things:

- create an action object
- post the action

To create the action object, you instantiate an object of the action class that corresponds to the user's command. You provide whatever data is necessary to properly initialize the action object.

After creating the object, call `PostAction()`. That's all there is to it. PowerPlant takes care of everything else.

When you call `PostAction()`, the action is executed. PowerPlant uses the `RedoSelf()` function to perform the initial command. PowerPlant then takes care of the menu item text. If the user chooses either the **Undo** or **Redo** commands, the undoer identifies the command and performs the appropriate action.

Implement Multilevel Undo

Each action object encapsulates all the information necessary to restore the application to a previous state. A series of action objects can be used to march backward or forward through the history of the user's actions.

PowerPlant does not have an automatic or standard method for implementing multilevel undo. You might go about accomplishing this goal in a variety of ways. Whatever approach you take, the `Finalize()` function in LAction might serve as a useful hook.

Because `Finalize()` is called after the user creates a new action object for an undoer, but before the previous action is deleted, you can make a copy of the action object and preserve it. This is the critical distinction between single-level undo (where the action is deleted when it is replaced) and a multilevel undo (where the action is preserved).

In a multilevel undo, you no longer toggle between two states and discard all previous actions. The undoer must serve as a means of traversing the entire action history.

You might create an LMultiUndoer attachment that maintains two stacks of action objects, one for undo and one for redo. When you do an action for the first time, you push it onto the undo stack. At the same time you would typically delete any items in the redo stack. By taking a new action, the user is abandoning any future redo of previously undone operations.

When you undo an action, you simply pop it from the undo stack, and push it onto the redo stack.

There are other complications. Rather than a single **Undo** item in the Edit menu, you need two items, one for **Undo** and one for **Redo**. As

an action is undone, the **Undo** item updates to reflect the current undo action, and the **Redo** item updates to reflect the current redo action.

Exactly what design you use and how you implement multilevel undo is up to you. However, your users will love you for it.

Summary of Undo in PowerPlant

Implementing undo in PowerPlant revolves around the LAction class. You declare and define an action class for each different kind of user action. The action object stores necessary state information. You create resources for the **Undo** and **Redo** menu item text. You attach an undoer object to each commander that handles an undoable action. When the user issues a command, you create the appropriate action object and post the action.

Use the `LAction::Finalize()` function as a hook to support multilevel undo.

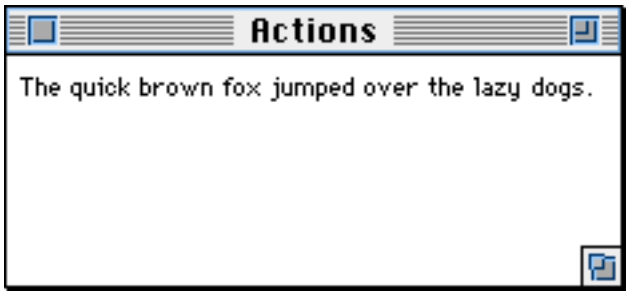
Code Exercise for Actions

In this exercise you create an application named "Actions." This exercise demonstrates in as simple a fashion as possible how to implement undo functionality in a PowerPlant application.

The code in this exercise builds on the code from Chapter 10, "Commanders and Menus" in *The PowerPlant Book*. You may wish to review that code exercise if you are unfamiliar with commanders.

When you run the Actions application, a window appears that contains some text, as shown in Figure 4.2. Remember that a caption simply displays text. You cannot edit the text in this window.

Figure 4.2 The Actions application in action



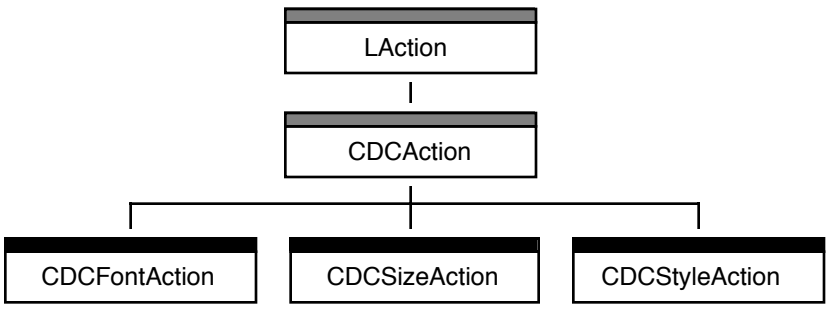
The application has the usual text-related menus, including **Font**, **Size**, and **Style**. In this exercise, you modify the behavior of the CDynamicCaptionCmdr object so that any menu command from these three menus can be undone or redone. In the process you will perform the three general tasks necessary to implement undo. In steps 1-5 you create an action class. Then you attach an undoer to the commander, and post an action in response to a menu command.

1. Examine the class declaration for action classes.

`CDynamicCaptionActions.h`

The `CDynamicCaptionActions.h` file declares all the action classes in this program. Figure 4.3 shows the class hierarchy. For simplicity, we'll refer to the CDynamicCaptionAction classes using the shorthand CDCAction.

Figure 4.3 CDynamicCaptionAction class hierarchy



The CDCAction class is abstract because it does not define the `UndoSelf()` and `RedoSelf()` functions. CDCAction has two data members:

- `mDynamicCaptionCmdr`—an LCommander
- `mDynamicCaption`—the caption object

In fact, these are the same object. However, storing a pointer to the caption object twice as different data types eliminates the need for typecasting later on.

The CDCAction class also defines `CanRedo()` and `CanUndo()` for all subclasses. Each function ensures that the commander is on duty before returning a value indicating that the action can be redone or undone.

Finally, you should take a look at the constructor for the CDCAction class. You don't have to write this code, it is provided for you. Here's the code for reference.

```
CDynamicCaptionAction::CDynamicCaptionAction(
  Int16                 inDescriptionIndex,
  LCommander         *inDynamicCaptionCmdr,
  CDynamicCaption    *inDynamicCaption,
  Boolean            inAlreadyDone )
    : LAction( rSTRx_RedoText,
      inDescriptionIndex, inAlreadyDone )
{
  // Save the commander and dynamic caption.
  mDynamicCaptionCmdr = inDynamicCaptionCmdr;
  mDynamicCaption = inDynamicCaption;
}
```

This code calls the LAction constructor and passes the resource ID number of the `'STR#'` resource containing the redo text for the **Edit** menu. It also passes the index number for the actual string in that resource to use, and a boolean value indicating that the task has already been done. In addition, the body of the function stores the pointer to the commander and the caption. In a subsequent step you will call this constructor yourself when you build a subclass of CDCAction.

The `CDynamicCaptionActions.h` file also declares three concrete action classes, one each for font, size, and style actions. In the rest of this exercise you implement the size action. The font and style actions are essentially the same, with minor differences to accommodate the data required to preserve state. In general, each concrete action class has data members to preserve state information, and defines the `RedoSelf()` and `UndoSelf()` functions.

2. Complete the class declaration for a size action.

`CDynamicCaptionSizeAction` `CDynamicCaptionActions.h`

The size action class must have two data members to preserve the previous and current font size. In addition, you must declare the `RedoSelf()` and `UndoSelf()` member functions. The necessary code is listed here.

```
protected:
  Int16   mSize;
  Int16   mSavedSize;

  virtual void RedoSelf();
  virtual void UndoSelf();
};
```

Make sure you add this code to the correct class declaration. Before closing the file, take a look at the class constructor. It receives three parameters: the new size, and the pointers to the commander and caption object. In the next step you write the constructor function.

3. Define the size action constructor.

`CDynamicCaptionSizeAction()` `CDynamicCaptionAction.cp`

To complete this constructor you need to perform three tasks. You must call the base class constructor, store the new size, and preserve the current size. In other words, you are saving state information for the new and previous font size.

a. Call the base class constructor.

In the initializer list, call `CDynamicCaptionAction()`. You must pass the index value for the redo size string. It is the constant `kSTRx_Size`. Also pass the pointers for the commander and the caption.

b. Store the new size.

In the body of the function, set the `mSize` data member to the value you receive in `inSize`.

c. Preserve the current size.

Use the `mDynamicCaption` data member, and send the caption a `GetSize()` message to get the current size. Store the result in `mSavedSize`. You have now preserved both the new size and the previous size in the action class's data members. The code for all the substeps is listed here.

```
CDynamicCaptionSizeAction::
                    CDynamicCaptionSizeAction(
  Int16 inSize,
  LCommander *inDynamicCaptionCmdr,
  CDynamicCaption *inDynamicCaption )
  : CDynamicCaptionAction( kSTRx_Size,
    inDynamicCaptionCmdr, inDynamicCaption )
{
  // Copy the size.
  mSize = inSize;

  // Get the current size.
  mSavedSize = mDynamicCaption->GetSize();
}
```

4. Undo a size action.

`CDynamicCaptionSizeAction::UndoSelf()`
`CDynamicCaptionAction.cp`

Make sure you modify the `UndoSelf()` function for the CDynamicCaptionSizeAction class. This file actually has three `UndoSelf()` functions, one for each concrete action class.

In the body of the function, simply send the caption object a `SetSize()` message. Pass the original size, stored in the `mSavedSize` data member.

```
mDynamicCaption->SetSize( mSavedSize );
```

5. Redo a size action.

`CDynamicCaptionSizeAction::RedoSelf()`
`CDynamicCaptionAction.cp`

Make sure you modify the `RedoSelf()` function for the CDynamicCaptionSizeAction class. In the body of the function, simply send the caption object a `SetSize()` message. Pass the new size, stored in the `mSize` data member.

```
mDynamicCaption->SetSize( mSize );
```

Remember, the redo action is also the "do" action when the action is first posted. That's why redo sets the new size.

You have now successfully completed the first and toughest chore, declaring and defining an action class. Note how this class preserves the necessary state information, and then uses that information to modify the state of the application as necessary when you undo and redo the action.

All that remains is to attach an undoer to the commander, and to post the action when the user issues a command. Each task requires one line of code.

6. Attach an undoer to the caption commander.

`ObeyCommand()` `CActionsApp.cp`

The undoer is an attachment. It should be attached to the caption commander. You create the commander when you create the window. After successfully creating the caption commander, send the caption an `AddAttachment()` message and add a new LUndoer object. The pointer to the caption commander is in a local variable, `theCaption`.

```
theWindow->SetLatentSub( theCaption );

// Add an undoer to the caption.
theCaption->AddAttachment( new LUndoer );

theWindow->Show();
```

7. Post a size action.

`ObeyCommand` `CDynamicCaptionCmdr.cp`

When the user chooses an item in the Size menu, post a size action. Call `PostAction()`. The action you are posting is a new CDynam-

icCaptionSizeAction. The new size the user chose is in `theSize`. Pass a pointer to the current object for the other two parameters required for the CDynamicCaptionSizeAction constructor.

```
::StringToNum( theMenuText, &theSize );

// Set the caption size.
PostAction( new CDynamicCaptionSizeAction(
                        theSize, this, this ) );
```

In the original commander defined in Chapter 10 of *The PowerPlant Book*, the commander did not post an action. Instead, it sent a `SetSize()` message directly to the caption (in this case, itself).

This simple fact highlights the difference between a strategy that supports undo, and a strategy that does not. The undo strategy imposes a minor indirection. Action-related information is created and preserved in the action object. Because the action object persists for a while, and because it stores the necessary state information, you can implement undo.

The font and style actions are effectively identical to the size action. The commander posts the appropriate action object when necessary. Each action object preserves the necessary state, and each implements the undo and redo behaviors by sending the appropriate message to the caption object.

8. Build and run the application.

When the application builds successfully and runs, the caption window appears as shown previously in Figure 4.2 on page 108. You

cannot change the content of the text, but you can make choices from the **Font**, **Size**, and **Style** menus.

Before you make a choice in any menu, examine the Edit menu. It is disabled, and the first item is **Can't Undo**. Now, make a choice in the **Size** menu, and then examine the **Edit** menu again.

The **Edit** menu is enabled, and the first item is **Undo Size**. Undo and redo the action and study what happens in the **Edit** menu. Change font and style, and see what happens.

In each case, you can undo and then redo the most recent action. The **Edit** menu updates appropriately.

Remember your most recent action in a window. Create a second window, and perform some actions in that window. Then switch back to the first window. The most recent action in that window is available for undo or redo. Each commander has its own undoer, and its own action. Not bad.

Congratulations! You have fully implemented single-level undo in an application. If you'd like to explore further, there are two areas ripe for improvement.

PowerPlant includes classes to support text actions such as cut, paste, and typing. Modify the application so that you can enter text, and add support for typing actions.

If you want a real challenge, implement multi-level undo. Read the hints in "Implement Multilevel Undo" on page 106. Good luck, and have fun!

5

Tables in PowerPlant

This chapter discusses how to display tabular data in a PowerPlant application.

Introduction to Tables in PowerPlant

Tabular data is a feature of many applications. The ability to display data in one or two dimensions is a common mechanism familiar to users. Tables are so common that it makes little sense to reinvent common table-related functionality, and that's where PowerPlant comes into play.

The PowerPlant table classes provide a framework for displaying tabular data. Taken together, the PowerPlant table-related classes provide all the basic functionality you expect from a table, and much more. You can display data in individual cells. You can add and remove rows and columns of cells. You can add, remove, and modify data in cells. You can display any kind of data: words, numbers, icons, pictures, and so forth. You can even create hierarchical tables where you can expand or collapse the display of sublevels of data to arbitrary depth.

These features mean that you can use the table classes as a powerful list manager. Just think of a list as a table with one column.

In this chapter we discuss how you can use table classes in a PowerPlant application. The topics discussed include:

- Table Strategy—the PowerPlant approach to tables
- Table Classes—the individual PowerPlant classes involved in tables
- Implementing Tables in PowerPlant—working with PowerPlant's table classes

Table Strategy

PowerPlant has gone through three iterations of support for tabular data. For information on the original LTable class, see *The PowerPlant Book*.

The classes that represent the second iteration for tabular data center around LNTable. These classes are located in the `PowerPlant:· In Progress:· AppleEvent Classes:Table & Text Classes:Table Classes` folder. In the future, they will move into the obsolete folder.

This chapter discusses the third and newest set of table classes. These classes are based on the LTableView class found in the `PowerPlant:Table Classes` folder. The LTableView design is highly modular for flexibility and extensibility. It factors table-related functionality into core table classes and three or four associated helper classes. Some of the support classes are found in the `PowerPlant:· In Progress:· Table Classes` folder.

With respect to the LTableView classes, in this section we discuss:

- Table Architecture—the overall design of the table classes
- General Table Implementation—issues related to how PowerPlant counts and manages cells in a table

Table Architecture

A table cell has a geometry (its dimensions), can be selected, and stores data. PowerPlant has families of helper classes to manage:

- geometry—the size of a cell
- cell selection—managing single and multi-cell selections
- data storage—the data stored in each cell

When you create a table, you associate an instance of each helper classes with the table. By substituting a different kind of helper class you can modify the behavior of the table.

For example, if you want every cell in the table to be the same physical size, you use LTableMonoGeometry as the geometry helper

class. If you want cells that can vary in size, you use LTableMultiGeometry. Likewise, if you want to allow only one cell to be selected at a time, LTableSingleSelector serves the purpose. If you want to allow multiple cells to be selected, LTableMultiSelector will do.

While this strategy creates a large number of classes, the underlying architecture is highly modular and flexible. By factoring out behavior that relates to cell size, cell selection, and data storage, you can use the same base table class for a variety of different kinds of displays, with different levels of functionality.

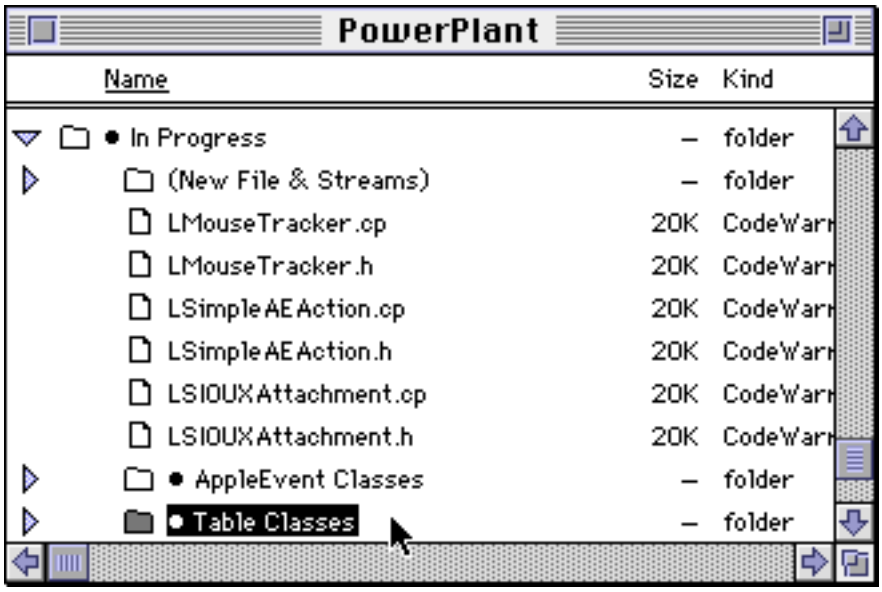
This design also gives you the opportunity to replace individual portions of table behavior with custom behavior designed to suit your own purpose.

There is one more kind of helper class that can be a real benefit to managing the display of tabular data, the collapsable tree.

PowerPlant supports hierarchical tables. In a hierarchical table, one row in the table serves as a header or "node" in a tree that can expand to reveal a sublevel of data, or collapse to hide all subsidiary data. A familiar example of this functionality can be seen in a Finder window using a list view. Each folder in the list is a node. By clicking the expansion triangle to the left of the folder, you open the folder to see its contents. The folder may contain other folders, and so on to arbitrary depth.

The PowerPlant hierarchical tables support this same functionality. You can have an expansion triangle appear to the left of a row in a hierarchical table. PowerPlant manages most of the details. It expands or collapses sublevels of data automatically, so you can concentrate on the real work.

Figure 5.1 Nodes in a hierarchical list



General Table Implementation

When working with tabular data, several questions can arise concerning how to address individual cells in a table. The answers to these questions are essentially arbitrary. However, knowing how to count cells is vital to success in managing tabular data.

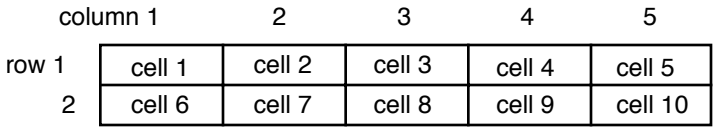
A row is a horizontal set of cells. A column is a vertical set of cells. PowerPlant uses 1-based counting for rows and columns in tables. The top row of a table is row 1, not row 0. The left column of a table is column 1, not column 0.

PowerPlant declares a data type, `TableIndexT`, for referring to columns, rows, and cells by index. This is an unsigned 32-bit integer. Tables in PowerPlant can have more than four billion cells.

Each cell in a table may be addressed by index number. Cells are ordered by column (across) first, and then by row (down). In other words, cell counting starts at the top left corner, then proceeds to the right across all columns along the row. When the full width of the

table has been reached, the count wraps to the beginning of the next row, as shown in Figure 5.2.

Figure 5.2 Counting cells in a table



The table classes have a function for getting the index number of any cell specified by row and column.

Finally, in PowerPlant a cell can be an object in its own right, an instance of the STableCell class. Details about this class can be found in "STableCell" on page 122. In brief, a cell object has a row and column, and a series of behaviors that allow easy manipulation of cell location (but not cell contents). Many of the member functions of the PowerPlant table classes require or return a reference to an STableCell object.

Table Classes

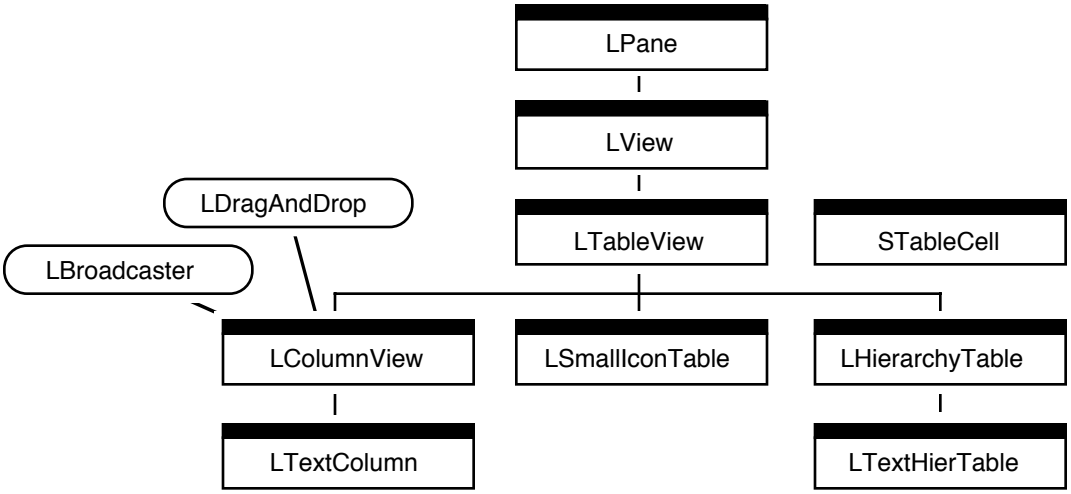
There are many classes related to tables in PowerPlant. This chapter does not discuss them all. This chapter concentrates on the latest and most powerful implementation of tables in PowerPlant.

For information on the LTable class, see *The PowerPlant Book* chapter on views. There is no formal documentation on the table classes found in the `Advanced Classes` folder. Read the source files if you're interested in these classes.

The classes discussed in this chapter are those based on and connected with LTableView. At the time of this writing, the source files for these classes are located in the PowerPlant `In Progress` folder.

These classes can be collected into two groups: the principal table classes, and the table helper classes. Figure 5.3 shows the table class hierarchy.

Figure 5.3 Table class hierarchy

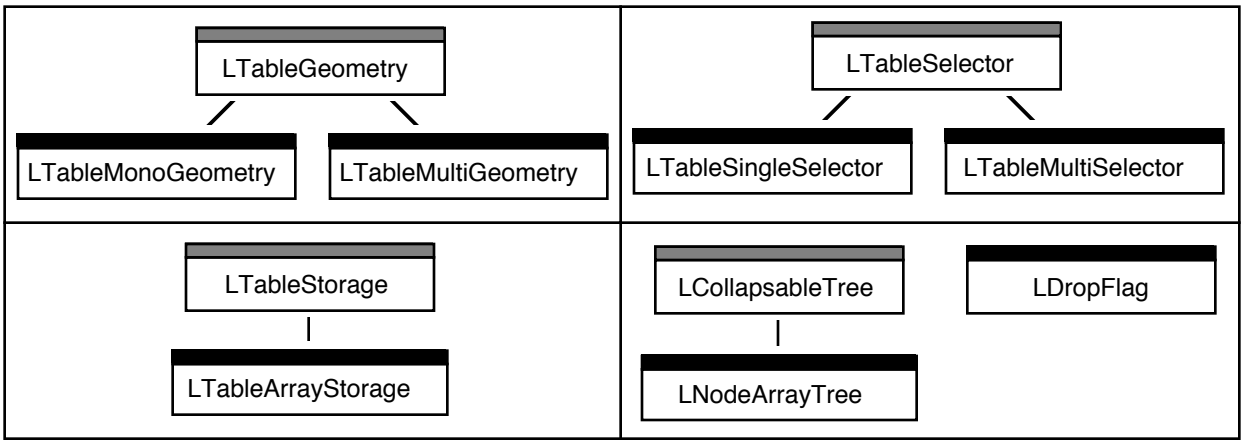


LTableView is the base table class. It inherits from LView. LColumnView supports a table with a single column. LTextColumn describes a table with one column of text items. LHierarchyTable supports nodes and multiple levels of data. LTextHierTable displays text in a hierarchical table. STableCell represents an individual cell in a table.

LSmallIconTable is a trivial implementation of LTableView to display small icons, and is intended to be a simple demonstration.

Figure 5.4 shows the four helper class hierarchies.

Figure 5.4 The table helper class hierarchies



The base class for each group of helper classes is an abstract class that defines the interface for the helper. Concrete classes provide various implementations.

LTableMonoGeometry provides a table where all cells are the same size. LTableMultiGeometry provides a table where cells may vary in size.

LTableSingleSelector provides a table where only one cell may be selected at any moment. LTableMultiSelector provides a table where multiple cells may be selected simultaneously, including discontiguous selection.

The LTableArrayStorage class provides for data storage for each cell in the table. You may use any subclass of LArray for data storage. You may choose LArray where each cell has the same length data storage. You may also use LVariableArray to support cells with varying data lengths.

The LNodeArrayTree class, working with LDropFlag, provides a standard implementation of a data hierarchy. The LDropFlag class manages the expansion triangle used to display or hide sublevels of data in a hierarchical table.

When you create a table, you instantiate a storage mechanism, a selector, and a geometry and attach them to the table. If the table is hierarchical, you also attach a tree helper object.

The remainder of this section discusses each table class in detail. The classes are:

- STableCell
- LTableView
- LColumnView
- LTextColumn
- LHierarchyTable
- LTextHierTable
- LSmallIconTable
- LCollapsableTree
- LNodeArrayTree
- LTableGeometry
- LTableMonoGeometry
- LTableMultiGeometry
- LTableSelector
- LTableSingleSelector
- LTableMultiSelector
- LTableStorage
- LTableArrayStorage
- LDropFlag

STableCell

STableCell is defined in the `UTables.h` file. All the member function definitions are in the header file, and are inline functions.

There are two data members, `row` and `col`. The constructor sets the cell location to (0,0) unless you provide a row and column. You can also pass a Point variable as an initializing value. PowerPlant sets `row` to the vertical value in the Point, and `col` to the horizontal value in the Point. Table 5.1 lists the member functions.

Table 5.1 STableCell member functions

Function	Purpose
`SetCell()`	set the cell's row and column
`IsNullCell()`	returns `true` if cell location is (0,0)
`ToPoint()`	convert cell location to a Point value

Function	Purpose
`operator ==()`	returns `true` if cell locations are the same
`operator !=()`	returns `true` if cell locations are not the same

In this context, cell location is the row and column location of the cell in the table.

When you create a table, you typically do not create a complete set of STableCell objects, one per cell. You create an STableCell object as necessary to manipulate an individual cell in the table. Also, note that none of the cell's member functions relate to the contents of the cell, only to cell location. You manipulate contents of the cell using member functions in the table classes.

LTableView

LTableView is a complex class that forms the basis for the table classes discussed in this chapter. The class data members store the data you need to create and manage a table. The class member functions implement standard table-related behavior.

LTableView derives from LPane via LView. Therefore it is a view like any other. See *The PowerPlant Book* chapters on panes and views for details of this aspect of LTableView.

Although LTableView is a concrete class, you do not typically instantiate an LTableView object. The default behavior of LTableView creates a table whose cells contain the row and column number. At the very least you would typically override the drawing routines to display the appropriate data for your table.

TIP: LTableView is an excellent class to instantiate for study purposes because it is a complete implementation of a default table (albeit with demonstration data). You can create an LTableView object, and then walk through the code as you perform table-related operations to see how PowerPlant works.

LTableView has five data members, detailed in Table 5.2.

Table 5.2 LTableView data members

Data member	Stores
`mRows`	number of rows in table
`mCols`	number of columns in table
`mTableGeometry`	pointer to the table geometry object
`mTableSelector`	pointer to the table selector object
`mTableStorage`	pointer to the table data storage

The number of rows and columns determines the dimensions of the table. The number of rows and columns is set after the table has been created.

The other data members correspond to the three helper classes associated with every table. (Hierarchical tables also have a tree helper object). Typically you create an LTableView descendant based on a PPob resource built in constructor. After the view is created, you instantiate helper objects and set the data members. The member functions for setting these data members are:

- `SetTableGeometry()`
- `SetTableSelector()`
- `SetTableStorage()`

LTableView Services

LTableView has member functions designed to provide a wide variety of services devoted to:

- Row, column, and cell management
- Accessing cells
- Cell geometry
- Cell selection
- Data storage

- Drawing and clicking

Row, column, and cell management

Table 5.3 lists some of the row, column, and cell management functions and their purpose.

Table 5.3 LTableView cell management functions

Function	Purpose
`GetTableSize()`	provides number of rows and columns
`IsValidRow()`	returns `true` if row exists in table
`IsValidCol()`	returns `true` if column exists in table
`IsValidCell()`	returns `true` if cell exists in table
`InsertRows()`	adds rows and data to the table
`InsertCols()`	adds columns and data to the table
`RemoveRows()`	removes rows and data from the table
`RemoveCols()`	removes columns and data from the table

There are no accessors to set the `mRows` and `mCols` data members directly. You should only modify those values by calls to the appropriate functions listed above.

Accessing cells

Table 5.4 lists some of the cell access functions and their purpose. Use these functions to find a desired cell in the table

Table 5.4 LTableView cell access functions

Function	Purpose
`CellToIndex()`	given a cell location, provides index for cell
`IndexToCell()`	given an index, provides location of cell

Function	Purpose
`GetNextCell()`	provides next cell
`GetNext-SelectedCell()`	provides next selected cell

Cell geometry

Table 5.5 lists some of the cell geometry functions and their purpose. Use these functions to get or modify cell size.

Table 5.5 LTableView cell geometry functions

Function	Purpose
`GetImageCell-Bounds()`	provides bounds of cell in image coordinates
`GetLocalCell-Rect()`	provides bounds of cell in local coordinates, returns `true` if cell is in frame (visible)
`GetRowHeight()`	returns the height of the specified row
`SetRowHeight()`	sets the height of the specified rows
`GetColWidth()`	returns the width of the specified column
`SetColWidth()`	sets the width of the specified columns

Each of these functions sends messages to the associated geometry helper object.

You cannot modify the size of an individual cell. You must modify the dimensions for an entire row or column. All cells in the same row must have the same height. All cells in the same column must have the same width. You may set the size of several contiguous rows or columns in a single call.

Cell selection

Table 5.6 lists some of the cell selection functions and their purpose. Use these functions to modify the selection range in a table.

Table 5.6 LTableView cell selection functions

Function	Purpose
`CellIsSelected()`	returns `true` if cell is selected
`SelectCell()`	adds cell to current selection
`SelectAllCells()`	selects all cells
`UnselectCell()`	removes cell from current selection
`UnselectAllCells()`	unselects all selected cells
`ClickSelect()`	adjusts selection in response to a click
`SelectionChanged()`	notification that selection has changed, empty function

Each of these functions (except `SelectionChanged()`, which is empty) sends messages to the associated selection helper object.

Data storage

Table 5.7 lists some of the data management functions and their purpose.

Table 5.7 LTableView data storage functions

Function	Purpose
`SetCellData()`	sets the data for the specified cell
`GetCellData()`	gets the data for the specified cell
`FindCellData()`	provides the cell that contains specified data

Each of these functions sends messages to the associated storage helper object.

When you get data, you must provide a pointer to storage that you have allocated. In PowerPlant, storage helpers copy data from the table into your storage. You do not get a pointer to the cell data stor-

age. Similarly, when you set data for a cell your data is copied into the cell storage.

WARNING! The calls to `GetCellData()` and `SetCellData()` require a reference to an `STableCell` to specify the cell. In the `STableCell` you specify the row and column of the cell. In a hierarchical table, those values must reflect the position the cell would occupy if the table were fully expanded. This is called the wide-open cell. For more on the concept of a wide-open table, see "LHierarchyTable" on page 130.

Drawing and clicking

Table 5.8 lists some of the drawing and clicking functions and their purpose. Use these functions to manage the visual appearance of the table and its cells, and to manage clicks in cells.

Table 5.8 LTableView drawing and clicking functions

Function	Purpose
`RefreshCell()`	redraw cell during next update event
`RefreshCellRange()`	redraw a range of cells during next update event
`HiliteCell-Actively()`	draw or undraw active highlighting for a cell
`HiliteCell-Inactively()`	draw or undraw inactive highlighting for a cell
`ActivateSelf()`	notification table is becoming active
`DeactivateSelf()`	notification table is becoming inactive
`DrawSelf()`	draw the table
`DrawCell()`	draw a specified cell
`ClickSelf()`	handle a click in the table

Function	Purpose
`ClickCell()`	handle a click in a particular cell
`GetCellHitBy()`	identify cell containing a point

LTableView and its descendants can properly handle both foreground and background highlighting. If the only thing your table does on activating or deactivating is modify highlighting, the default functions take care of you. If you want to add functionality, you can override `HiliteCellActively` and `HiliteCellInactively`.

The `DrawSelf()` and `ClickSelf()` functions search for the cell(s) involved and call `DrawCell()` or `ClickCell()`.

You will override the `DrawCell()` and `ClickCell()` functions in derived classes. The default `DrawCell()` function in LTableView draws a string in the cell that contains the row and column number. The `ClickCell()` function beeps on a double-click. You must replace this behavior with functionality appropriate for the kind of data you display in your table.

LColumnView

LColumnView is a simple class that derives from LTableView. It also inherits from LDragAndDrop and LBroadcaster. The intent of this class is to define an interface for a table that contains a single column of data. Functions for adding and removing columns have been overridden to do nothing but display a signal if signaling is on.

This class supports drag and drop in the table cells. It broadcasts a message when a cell is double-clicked, or when the selection changes.

LColumnView does not override `LTableView::DrawCell()`, and so cannot be used directly. In typical use, you would subclass LColumnView to display data of the appropriate type. The LTextColumn class does that.

LTextColumn

LTextColumn is a simple class that derives from LColumnView. The intent of this class is to create a table that consists of a single column of text. This is a common visual interface object: a simple list of text items.

LTextColumn has a single text traits resource that applies to all cells in the table. It uses a 'STR#' resource to specify the initial items in the column.

WARNING! The constructor reads the data from the 'STR#' resource, but does not release the resource. You should mark the resource purgeable if you wish to free up the memory.

You can use functions inherited from LTableView to modify the contents of the cells after creation. The only function LTextColumn overrides is `LTableView::DrawCell()`.

LSmallIconTable

LSmallIconTable is a simple class derived from LTableView. This class serves as a demonstration of how to derive a class from LTableView to display a unique kind of data—in this case, a small icon.

This class uses the LTableMonoGeometry, LTableSingleSelector, and LTableArrayStorage helper classes to create a table with cells of a uniform size and that allows one cell to be selected.

This class has no additional data members or member functions. It does override `DrawCell()` to plot an icon in the cell. It stores the small icon ID and a name for each icon in a simple struct, which serves as the data for each cell.

LHierarchyTable

LHierarchyTable is a moderately complex class that adds the ability to display tabular data hierarchically to LTableView. LHierarchyTable adds two new data members, listed in Table 5.9.

Table 5.9 LHierarchyTable data members

Data member	Stores
`mCollapsableTree`	pointer to the tree helper object
`mFlagRect`	size of the drop flag

The `mCollapsableTree` member is analogous to the geometry, selection, and storage data members in LTableView. The table object uses this value to access the services of the tree helper object.

The LHierarchyTable class uses an LNodeArrayTree as its tree helper object. The LHierarchyTable constructor function creates an LNodeArrayTree object for the table. See "LNodeArrayTree" on page 141.

The `mFlagRect` data member contains the size of the standard expansion triangle (the drop flag). The `LHierarchyTable::ClickSelf()` member function uses this rectangle to determine if a click is in the expansion triangle or not, and responds accordingly.

Table 5.10 lists several of the LHierarchyTable member functions. You should not have to modify any of these functions. However, use these functions to get the correct index number and to add rows to the table at the proper level in the hierarchy.

Table 5.10 LHierarchyTable member functions

Function	Purpose
`GetWideOpenTableSize()`	provides number of rows and columns in a fully expanded table
`GetWideOpenIndex()`	translates exposed index into wide open index
`GetExposedIndex()`	translates wide open index into exposed index
`InsertSiblingRows()`	add rows as siblings to row at the insertion point

Function	Purpose
`InsertChildRows()`	add rows as children to row at the insertion point
`AddLastChildRow()`	add one row as the last child of the specified parent row

A hierarchical table may be partially or fully expanded. You may need to get the index number of a cell in either situation. Use `GetWideOpenIndex()` and `GetExposedIndex()` to convert an index from one form to the other. Expanding and collapsing parts of the table only affects rows. There is no way to hide columns of data.

When you insert a row or rows into a table, you specify the row after which the new rows should appear. In a hierarchical table, the new rows may be at the same level as the "after" row (a sibling), or nested inside the "after" row (a child).

WARNING! The calls to insert child or sibling rows require that you specify the wide-open index number of the row after which you want the new rows to appear.

LHierarchyTable overrides `InsertRows()` so that it creates sibling rows. This mimics the behavior of a non-hierarchical table. LHierarchyTable also overrides `RemoveRows()`. You can only remove one row at a time from a hierarchy table. If that row is a parent row (one with children), all the children are removed as well.

LHierarchyTable also has functions that handle expanding and collapsing levels in the hierarchy. If you click on the expansion triangle, you expand or collapse that level. If you Option-click the expansion triangle, you expand or collapse all levels below that level (known as deep expand or deep collapse).

In a typical implementation of a hierarchical table you won't have to call or override the member functions concerned with expanding or collapsing levels in the hierarchy. PowerPlant takes care of all the housekeeping for you.

LTextHierTable

LTextHierTable is a simple class derived from LHierarchyTable. This class serves as a demonstration of how to derive a class from LHierarchyTable to display text.

This class adds four data members, detailed in Table 5.11

Table 5.11 LTextHierTable data members

Data member	Stores
`mLeafTextTraits`	text traits for a child row
`mParentTextTraits`	text traits for a parent row
`mFirstIndent`	indent for first level text cell
`mLevelIndent`	indent per additional level

This class uses the text traits data members to control the appearance of text. By default, they are the system font for child rows, and the application font for parent rows.

This class adds no new member functions. It overrides `DrawCell()`, `HiliteCellActively()`, and `HiliteCellInactively()` to draw and highlight text without including the expansion triangle.

LTableGeometry

LTableGeometry is an abstract class that specifies the interface for the geometry helper objects. These functions provide behaviors to maintain the location, width, and height of each cell in a table.

This class has one data member, `mTableView`. This is a pointer to the table that owns this helper.

All of the functions in this class are pure virtual or empty. These functions form the basis for interacting with a table's geometry in PowerPlant. Most of the time you will not need to concern yourself

with these functions. The PowerPlant table classes call these functions to get or set required data. In general, you should call the table functions, not the related geometry functions.

For background purposes, Table 5.12 lists LTableGeometry functions.

Table 5.12 LTableGeometry member functions

Function	Purpose
`GetImageCell-Bounds()`	provides bounds of cell in image coordinates
`GetRowHitBy()`	returns index of row that contains a point
`GetColHitBy()`	returns index of column that contains a point
`GetTableDimensions()`	provides size of table in pixels based on number of rows and columns
`GetRowHeight()`	return height of specified row
`SetRowHeight()`	set height of specified row(s)
`GetColWidth()`	get width of specified column
`SetColWidth()`	set width of specified column(s)
`InsertRows()`	add row(s) after specified row
`InsertCols()`	add column(s) after specified column
`RemoveRows()`	remove specified row(s)
`RemoveCols()`	remove specified column(s)

All the get and set functions are pure virtual.

The insert and remove functions are all empty. The purpose of these functions in subclasses is to maintain the geometry, not to actually add or remove cells from the table. See LTableMultiGeometry for an example.

LTableMonoGeometry

LTableMonoGeometry is a concrete implementation of LTableGeometry to support a table where all cells are the same size.

This class has two data members, `mColWidth` and `mRowHeight`.

LTableMonoGeometry adds no new member functions, but implements every pure virtual function listed in LTableGeometry.

The insert and remove rows and columns functions remain empty.

LTableMultiGeometry

LTableMultiGeometry is a concrete implementation of LTableGeometry to support a table where rows and columns may vary in size.

This class adds four data members, as detailed in Table 5.13.

Table 5.13 LTableMultiGeometry data members

Data member	Stores
`mRowHeights`	array of heights for rows in table
`mColWidths`	array of widths for columns in table
`mDefaultRowHeight`	height of new rows
`mDefaultColWidth`	width of new columns

LTableMultiGeometry adds no new member functions, but implements every function listed in LTableGeometry.

The insert and remove rows and columns functions maintain the arrays of heights and widths for the table.

LTableSelector

LTableSelector is an abstract class that specifies the interface for the selector helper objects. These functions provide behaviors to maintain the selection range in a table.

This class has one data member, `mTableView`. This is a pointer to the table that owns this helper.

All of the functions in this class are pure virtual or empty. These functions form the basis for interacting with selected cells in PowerPlant. Most of the time you will not need to concern yourself with these functions. The PowerPlant table classes call these functions to get or set required data. In general, you should call the table functions, not the related selector functions.

For background purposes, Table 5.14 lists LTableSelector functions.

Table 5.14 LTableSelector member functions

Function	Purpose
`CellIsSelected()`	returns true if cell is selected
`SelectCell()`	adds cell to current selection
`SelectAllCells()`	selects all cells
`UnselectCell()`	removes cell from current selection
`UnselectAll-Cells()`	deselects all cells
`ClickSelect()`	adjust selection when clicking on a cell
`DragSelect()`	adjust selection while user drags
`InsertRows()`	add row(s) after specified row
`InsertCols()`	add column(s) after specified column
`RemoveRows()`	remove specified row(s)
`RemoveCols()`	remove specified column(s)

All the select functions are pure virtual.

The insert and remove functions are all empty. The purpose of these functions in subclasses is to maintain the selection range, not to actually add or remove cells from the table.

LTableSingleSelector

LTableSingleSelector is a concrete implementation of LTableSelector for a table that may have one and only one cell selected at a time.

This class has one data member, `mSelectedCell`. This is an STableCell object representing the currently selected cell, if any.

LTableSingleSelector adds no new member functions, but implements every function listed in LTableSelector.

LTableMultiSelector

LTableMultiSelector is a concrete implementation of LTableSelector to support a table where multiple cells may be selected. LTableMultiSelector supports discontiguous selection.

This class adds two data members: `mSelectionRgn` and `mAnchorCell`. The anchor cell is the most recently selected cell. The selection region describes the "region" occupied by selected cells.

The use of the region in this instance is actually a neat trick. The code that selects a cell adds a square 1-pixel in size to the region. That square is defined by the cell's row and column as the top left coordinate, and adds one pixel for the bottom right coordinate. Then, to determine if a cell is selected, the code simply checks whether that row and column falls within the selection region.

LTableMultiSelector adds one new member functions, `SelectCellBlock()` to select a range of cells.

TIP: If you wish to operate on all selected cells, use the LTableView function `GetNextSelectedCell()` to walk through the entire table, stopping on selected cells.

LTableStorage

LTableStorage is an abstract class that specifies the interface for the data storage helper objects. These functions provide behaviors to maintain the data associated with a table.

This class has one data member, `mTableView`. This is a pointer to the table that owns this helper.

All of the functions in this class are pure virtual. These functions form the basis for interacting with data in the table. Most of the time you will not need to concern yourself with these functions. The PowerPlant table classes call these functions to get or set required data. In general, you should call the table functions, not the related storage functions.

For background purposes, Table 5.15 lists LTableStorage functions.

Table 5.15 LTableStorage member functions

Function	Purpose
`SetCellData()`	set data for an individual cell
`GetCellData()`	copy data from an individual cell
`FindCellData()`	search cells for specified data
`GetStorageSize()`	provides the number of columns and rows for which data is stored
`InsertRows()`	add data for specified rows
`InsertCols()`	add data for specified columns
`RemoveRows()`	remove data for specified row(s)
`RemoveCols()`	remove data for specified column(s)

The purpose of the insert and remove functions in subclasses is to add or remove data, not to add or remove cells from the table.

When inserting new rows and columns, the data for a single cell is specified. All new cells receive the same data.

LTableArrayStorage

LTableArrayStorage is a concrete implementation of LTableStorage to support a table where data is stored in an array (an LArray object, or an object of a class that inherits from LArray).

This class has two data members, `mDataArray` and `mOwnsArray`. The `mDataArray` member is a pointer to the array that holds the data. The `mOwnsArray` member determines whether the array is destroyed when the LTableArrayStorage object is destroyed.

The nature of the array can be specified using various LTableArrayStorage constructors. There are three constructors. The parameters and purpose of each are listed in Table 5.16.

Table 5.16 LTableArrayStorage constructors

Parameters	Purpose
`(LTableView*, Uint32)`	for cells with same size data, creates an LArray object with items equal to size provided
`(LTableView*)`	for cells with varying size data, creates an LVariableArray
`(LTableView*, LArray*)`	user-specified subclass of LArray

LTableArrayStorage adds no new member functions, but implements every function listed in LTableStorage.

LCollapsableTree

LCollapsableTree is an abstract class that specifies the behavior of a hierarchical tree with expandable nodes.

This class has no data members.

All of the functions in this class are pure virtual. These functions form the basis for interacting with a hierarchical tree. Most of the time you will not need to concern yourself with these functions. The PowerPlant hierarchy table classes call these functions to get or set required data. In general, you should call the table functions, not the related LCollapsableTree functions.

For background purposes, Table 5.15 lists some LCollapsableTree functions.

Table 5.17 Some LCollapsableTree member functions

Function	Purpose
`GetWideOpenIndex()`	translates exposed index into wide open index
`GetExposedIndex()`	translates wide open index into exposed index
`GetParentIndex()`	gets index of parent row
`GetNestingLevel()`	gets nesting depth of specified row
`InsertSibling-Nodes()`	add nodes as siblings to the node at the insertion point
`InsertChildNodes()`	add nodes as children to the node at the insertion point
`AddLastChildNode()`	add one node as the last child of the specified parent row
`RemoveNode()`	delete a node and its descendants

There is no function in the table classes that corresponds to the `GetNestingLevel()` function. If nesting level is a concern, you can send the message directly to the tree helper object through the `mCollapsableTree` data member of the table. See "Drawing a Cell" on page 149.

There are also functions to expand and collapse nodes, and to perform other node-related functions.

LNodeArrayTree

LNodeArrayTree is a concrete implementation of LCollapsableTree. It uses an array to track the hierarchy of rows and nested rows.

This class has two data members, `mHierarchyArray` and `mExposedNodes`. The `mHierarchyArray` member is a pointer to the array that holds the nested hierarchy of nodes. The `mExposedNodes` member is the number of exposed nodes at any moment.

LDropFlag

LDropFlag manages the expansion triangle in a hierarchical table. It has two static functions, `Draw()` and `TrackClick()`. The hierarchy table classes use these functions to draw the expansion triangle and to determine whether the user has clicked in the triangle.

WARNING! To use LDropFlag you must also include the `DropFlags Icons.rsrc` file in your project.

Implementing Tables in PowerPlant

This section discusses how to work with the PowerPlant table classes from a task-based perspective. While the collection of table-related classes appears very complex, in fact they hide an underlying elegance in design. Using these classes is actually a straightforward process. The specific table classes, LTableView and LHierarchyTable, provide almost all the functionality you need.

You will occasionally direct messages to helper objects, but by and large the complexity of the geometry, selection, storage, and tree classes is hidden.

Each topic reflects a table-related task you must perform. This section includes the following topics:

- Creating a Table
- Managing Rows and Columns
- Setting Cell Data
- Getting Cell Data
- Handling Clicks in a Cell
- Responding to Selections
- Drawing a Cell
- Finding Cells
- Finding Data in a Table
- Scrolling a Table

Creating a Table

The simplest way to create a table is to use Constructor when you generate the visual interface for the window containing the table. Use an LTableView object or one of its descendants (either a standard PowerPlant class or a custom class you created). See the *Constructor Manual* for details.

A table built in this way does not have any of the necessary helper objects, not does it have any cells or data.

Creating helper objects

You need to create three helper objects and attach them to the table after the table object is instantiated. They are the geometry helper, the selection helper, and the storage helper. Create objects of the class you choose for the functionality you desire—uniform cell size or not, single or multiple cell selection, the type or array for storage.

You can create these helper objects in the table constructor function, in the table's `FinishCreateSelf()` function, or in an initializer function called from the table constructor. Use `operator new` and specify the appropriate class constructor for each helper. Store the pointer to the objects in the table's `mTableGeometry`, `mTableSelector`, and `mTableStorage` data members, respectively.

Adding rows and columns

After creating the helper objects, you need to insert the appropriate number of rows and columns. See "Managing Rows and Columns" for details. The order of events is important. The process of adding rows and columns may affect the geometry, selection, and storage objects, so those objects should exist and be attached to the table before adding rows and columns.

You should also initialize the contents of each cell as necessary. See "Setting Cell Data" on page 146.

Managing Rows and Columns

Every table has rows and columns of cells. There are three things you can do with rows and columns. You can insert them into the table, remove them from the table, or manage their size.

Inserting columns

Use the `InsertCols()` function. It has five parameters.

Table 5.18 Parameters for `InsertCols()`

Data type	Parameter	Purpose
`Uint32`	`inHowMany`	number of columns to add
`TableIndexT`	`inAfterCol`	row after which new columns appear
`void *`	`inDataPtr`	pointer to data put in a cell

Data type	Parameter	Purpose
`Uint32`	`inDataSize`	number of bytes of data
`Boolean`	`refresh`	whether to refresh the table

The `inDataPtr` parameter points to one cell's data. Each cell receives the identical data initially. You can then set each cell's data as necessary. See "Setting Cell Data" on page 146.

Inserting rows

The `InsertRows()` function matches `InsertCols()` described in Table 5.18. You specify the row after which new rows appear.

WARNING! For hierarchical tables, the `inAfterRow` parameter in the call to `InsertRows()` must specify a wide-open index value. That is, it is the index number the row would have if the table were fully expanded.

For hierarchical tables, a call to `InsertRows()` creates non-collapsable sibling rows. You can also call `InsertSiblingRows()`. It has one additional parameter that specifies whether the row is collapsable or not. If you want to create a collapsable row, you must use `InsertSiblingRows()`.

Table 5.19 Parameters for `InsertSiblingRows()`

Data type	Parameter	Purpose
`Uint32`	`inHowMany`	number of rows to add
`TableIndexT`	`inAfterRow`	row after which new rows appear
`void *`	`inDataPtr`	pointer to data put in a cell
`Uint32`	`inDataSize`	number of bytes of data

Data type	Parameter	Purpose
`Boolean`	`inCollapsable`	whether rows are collapsable
`Boolean`	`refresh`	whether to refresh the table

The `inDataPtr` parameter points to one cell's data. If you are inserting more than one cell, each cell receives the identical data initially. You can then set each cell's data as necessary. See "Setting Cell Data" on page 146.

To insert child rows under a row, call `InsertChildRows()`. The parameters are the same as those for `InsertSiblingRows()`, except that the second parameter specifies the parent row under which the new child rows will appear.

Removing rows and columns

Call the LTableView functions `RemoveRows()` and `RemoveCols()`. Each call has three parameters, detailed in Table 5.20.

Table 5.20 Parameters for removing rows and columns

Data type	Parameter	Purpose
`Uint32`	`inHowMany`	rows or columns to remove
`TableIndexT`	`inFromRow` `inFromCol`	row or column after which rows and columns are removed
`Boolean`	`refresh`	whether to refresh the table

Removing a row or column also removes the associated data from the table storage.

You can only remove one row at a time from a hierarchical table. However, if that row is a parent row, all of its children are removed along with it.

Changing row and column size

Call the LTableView functions `SetRowHeight()` and `SetColWidth()`. Each call has three parameters, detailed in Table 5.21

Table 5.21 Parameters for changing row and column size

Data type	Parameter	Purpose
`Uint16`	`inHeight` `inWidth`	the new height or width
`TableIndexT`	`inFromRow` `inFromCol`	first row or column to have the new size
`TableIndexT`	`inToRow` `inToCol`	last row or column to have the new size

For tables that use LTableMonoGeometry, the range of rows or columns is ignored. All rows or columns are set to the new size.

For tables that use LTableMultiGeometry, the range specified is inclusive. The rows or columns at the beginning and end of the range are resized, along with all rows or columns in between.

Use `GetRowHeight()` or `GetColWidth()` to get the size of an individual row or column.

Setting Cell Data

A table usually has associated data storage. Storage is not mandatory. For example, the default implementation in LTableView has no storage. It draws the row and column number directly in each cell.

Use `SetTableStorage()` to provide a pointer to an LTableStorage helper object to the table.

If there is storage, then you can set each cell's data individually using the LTableView function `SetCellData()`. This function takes care of all the interaction between you and the data storage. It

determines where in the storage to place the data you provide. There are three parameters, as detailed in Table 5.22

Table 5.22 Parameters for setting cell data

Data type	Parameter	Purpose
`STableCell`	`inCell`	the wide-open cell
`void*`	`inDataPtr`	pointer to data put in a cell
`Uint32`	`inDataSize`	number of bytes of data

You specify the cell by row and column number in the `inCell` parameter. PowerPlant copies the data into storage. You can dispose of the original data after the call returns if you wish.

Getting Cell Data

It is frequently necessary to retrieve data associated with a cell. If there is storage, then you can get each cell's data individually using the LTableView function `GetCellData()`. This function takes care of all the interaction between you and the data storage. It determines where in the storage your data is located, and retrieves a copy of it for you. The three parameters are detailed in Table 5.22

Table 5.23 Parameters for getting cell data

Data type	Parameter	Purpose
`STableCell`	`inCell`	the cell in question
`void*`	`outDataPtr`	pointer to data buffer
`Uint32`	`ioDataSize`	number of bytes of data

You specify the row and column number of the cell in the `inCell` parameter.

You must allocate the data buffer before making this call. The buffer must be large enough to hold the data. You provide the size of the

buffer in the `ioDataSize` parameter. However, if you use an LArray object in the LTableArrayStorage (as opposed to LVariableArray), LArray ignores this parameter. LArray copies data that is the size of each item in the array *regardless of the size of the buffer*.

WARNING! If the data buffer you provide is smaller than the size of an item in an LArray attached to LTableArrayStorage, PowerPlant will raise a signal. Ignoring this problem can lead to crashes.

PowerPlant copies the data from storage and places it in the buffer, thus giving you a copy of the data, not a pointer to the data in storage. If the data for this cell changes in storage, your copy will be outdated until you get the data again.

TIP: Actually, there is a way you can access table data directly. Create an array object ahead of time, and keep a pointer to the object. Use that object when you create the LTableArrayStorage object that you attach to the table. You then have two paths into the table storage: the LTableView calls, and LArray calls such as `GetItemPtr()`. With a pointer to the array item, you can modify table storage directly. This may be optimal when you have continuously-updated data and speed is an issue.

Handling Clicks in a Cell

In most circumstances, when the user clicks in a cell you respond to the click. The `LTableView::ClickSelf()` function handles all cell selection automatically using the selector helper object you attach to the table. `LHierarchyTable::ClickSelf()` also handles expanding and collapsing levels in a hierarchical table.

You may want to implement additional behavior beyond simply selecting the cell. You may want to allow the user to select part of the contents of a cell (such as a range of text). Exactly how you implement these additional features is application dependent and beyond the scope of an application framework.

However, the framework does provide the hook. Declare a subclass of the appropriate table class (LTableView or LHierarchyTable). Override the `ClickCell()` function. PowerPlant calls this function whenever a click occurs inside a cell.

You can use `GetClickCount()` (an LPane function) to determine if a click is a single or multiple click.

Responding to Selections

If you wish to perform some action when the selection range in a table changes, override the `SelectionChanged()` function. The selection could change because a new cell is selected, a cell is added to an extended selection, a cell is removed from the selection range, selected cells are removed from a table, and so on. Whatever the cause, PowerPlant calls `SelectionChanged()` whenever the selection range changes.

What you do is, of course, up to you. You might want to walk the cells and perform some action based on which cell or cells are currently selected. You might want to update menus based on whether cells are selected. PowerPlant provides the hook. You provide the application-specific functionality.

Drawing a Cell

There are several issues that arise when it comes time to draw a cell. Of course, you must draw the contents. In addition, you must occasionally refresh the contents explicitly, or highlight the cell.

Drawing a cell

Precisely what you do to draw the data in your cell is, of course, data dependent. PowerPlant includes two example classes to demonstrate how it's done—LSmallIconTable (for LTableView) and LTextHierTable (for LHierarchyTable). The code exercise in this chapter implements `DrawCell()` to draw both an icon and text.

PowerPlant calls `DrawCell()` whenever you should render the contents of the cell. You override this function in your own table

class, and provide the necessary code to draw the data in the cell. Typically you perform some pixel-based calculations to determine precisely where the data should appear in the cell, and then use Mac OS Toolbox calls to draw the data.

In a hierarchical table, the nesting level can affect where you draw the data.

Nesting level

The nesting level is the number of levels down from the top level a particular cell occupies. Top level nodes are at nesting level zero.

If you stagger your data according to nesting level, then you must take a row's level into account before you draw data. This is one case where you address a helper object directly. Use the table's `mCollapsableTree` data member and send the tree helper object a `GetNestingLevel()` message. This call returns the nesting level. You can then use the nesting level and a standard indent of some amount to adjust the horizontal location of the information you draw in the cell.

The code exercise in this chapter demonstrates this technique.

The Drop Flag

In a hierarchical table, the first cell in an expandable row should have an expansion triangle. To display this flag, call `DrawDropFlag()`. You provide the cell, and the wide-open index value for the row. PowerPlant handles the rest.

Highlighting a cell

PowerPlant usually takes care of highlighting automatically. However, you may wish to control highlighting. For example, you should not highlight the drop flag area in a hierarchical table.

To modify highlighting behavior, override `HiliteCellActively()` and `HiliteCellInactively()`. You should not need to override `HiliteCell()` or `HiliteSelection()`.

WARNING! At the time of this writing, `LTableView::HiliteSelection()` has two separate implementations, one of which is inactive. The inactive code calls `HiliteCellActively()` and `HiliteCellInactively()`. The actual code handles highlighting directly, and therefore overrides of these two functions will not be called! Until the code stabilizes, you may need to override `HiliteSelection()` directly. Examine the source code for further enlightenment.

Refreshing a cell

PowerPlant usually handles updating the screen for you. However, there may be times when you want to explicitly mark a cell or range of cells for refreshing during the next update event. Use `RefreshCell()` or `RefreshCellRange()` for this purpose. You specify the cell or range of cells to refresh.

Finding Cells

Managing tabular data frequently requires that you walk through each cell, a range of cells, or each selected cell, in a table. As you go through the cells, you perform some operation on each cell.

PowerPlant provides two functions for walking the cells in a table. They are `GetNextCell()` and `GetNextSelectedCell()`. Each begins the walk at the cell you specify. Each returns a boolean value `false` when there is no next cell. You can use calls to these functions in a `while` loop to walk through the desired cells.

Cells are ordered by column (across), and then by row (down). Row zero is before the first cell. The next cell after row zero and any column is cell (1,1). Column zero is before column one. The next cell after row "r" and column zero is Cell (r,1).

To look for all cells in a table, you would write code like this:

```
STableCell theCell (0,0); // start with first cell
while (GetNextCell (theCell))
{
```

```
  // operate on cell
}
```

What you do when you find the cell is up to you. You can set or retrieve data, select or deselect the cell, search for data, and so forth.

Finding Data in a Table

PowerPlant provides a search mechanism for locating a cell that contains specified data. Call `FindCellData()`. This call has three parameters, detailed in Table 5.24

Table 5.24 Parameters for finding cell data

Data type	Parameter	Purpose
`STableCell`	`outCell`	cell that contains the data
`void *`	`inDataPtr`	pointer to data to look for
`Uint32`	`inDataSize`	number of bytes of data

You specify the data to search for and the length of the data. If a cell is found that contains the data, the call returns `true` and puts the cell location in `outCell`. Otherwise the call returns `false`.

`FindCellData()` always begins the search at the first cell in the table, and returns the first cell it encounters that contains the specified data. If you wish to search for multiple hits, you can't use `FindCellData()`. You'll have to write code to walk the cells and look in each cell's data individually. You could also create a custom data storage class with a different search mechanism.

Using the LTableArrayStorage class, storage is an array where each element in the array matches the corresponding index value of a cell in the table. The LTableArrayStorage uses an LArray function to search for the data in the array. The search goes from the beginning of the array to the end. The search terminates as soon as a matching data item is found. If you use a custom data storage class, the results

of `FindCellData()` will depend upon your implementation of the search.

Scrolling a Table

Tables do not have built in scroll bars. If the table image area is larger than the table frame, you should embed the table inside an LScroller or LActiveScroller view. See *The PowerPlant Book* chapter on views for information on scrolling.

Summary of Tables in PowerPlant

A table is a familiar mechanism for displaying lists or matrices of data in rectangular cells arranged by row and column. PowerPlant implements a well-factored, and elegant representation of tabular data. Because each table uses helper objects for cell geometry, cell selection, and data storage, it is possible to mix and match from a variety of options to create a table that has the functionality you need.

The resulting collection of table classes therefore appears complex. However, with a few exceptions, every function you need to use and understand is in either the LTableView or LHierarchyTable classes.

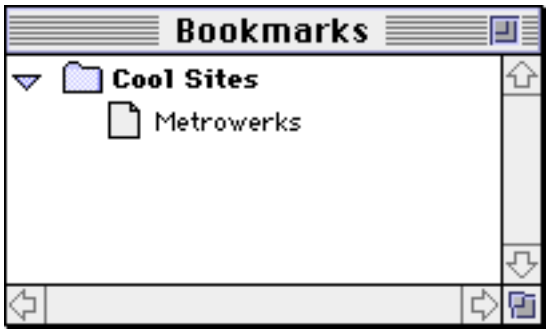
PowerPlant's ready-made helper classes for geometry, selection, storage, and trees provide most of the table-related functionality you need indirectly. LTableView and LHierarchyTable use these helper classes extensively, but the complexity in those classes is hidden from you, the PowerPlant programmer.

Actually creating and using a table in PowerPlant turns out to be fairly straightforward. There are specific functions to manage rows, columns, and cells. You can set or get data, draw data, handle clicks and cell selection, walk through cells in the table, and search for data. You perform many of these tasks in the code exercise accompanying this chapter.

Code Exercise for Tables

In this exercise you create an application that displays a hierarchical table. The table groups a list of items, in this case URL bookmarks. This isn't a full-featured application. It doesn't save data, open files, or support copy and paste. However, it does show you how to create, manage, and use a table in PowerPlant.

Figure 5.5 The bookmarks table



The table is one column wide with an arbitrary number of rows. There are two types of items that can occupy a cell, a group item and a data item. Each item displays a small icon and a label.

A group item is like a folder that contains either data items or other group items. You can expand and collapse a group item. In other words, the group item is a parent and can have children.

A data item cannot be expanded, because a data item cannot have children. Each data item is, in theory, a URL to some location on the World Wide Web. This particular little application does not have a mechanism for connecting to the web site. However, the potential for that functionality is built into the design, as you'll see when you go through the code exercise.

The necessary PPob resource has been built for you. It describes a standard window containing two panes. One is a scrolling view to contain the table. The other is a CBookmarksTable. CBookmarksTable inherits from LHierarchyTable to support expandable groups.

With this brief overview behind us, let's get into the code.

1. Examine the SBookmarkItem.

`Struct declaration` `CBookmarksTable.h`

The purpose of this step is to give you a little background into the kinds of data you will put into cells in this table. This has nothing to do with tables in general, but will help you understand what's going on in this exercise.

At the end of the CBookmarksTable header file there is a short struct declaration for an object of type STableItem. This struct represents an item in the table—that is, the contents of a single cell.

Here's the code for quick reference. You do not need to enter this code. It already exists and has been provided for you. As usual, existing code is shown in italic style.

```
struct STableItem {
  DataIDT mType;
  Str31   mName;
  Str255  mLocation;

  // Some constructors to make things easier.
  STableItem();
  STableItem( DataIDT inType );
  STableItem( DataIDT inType, Str31 inName,
              Str255 inLocation );
};
// Bookmark table item types.
const DataIDT  kGroupItemType = 'Grup';
const DataIDT  kBookmarkItemType = 'Book';
```

The same struct serves for both group and bookmark items. Each item has a type, a name, and a location. The name is the text that appears in the cell. The location is the URL associated with the item. For a group item, the `mLocation` field is typically empty.

In subsequent steps you will install items of each type as the data associated with cells in the table. All the remaining steps in this code exercise take place in `CBookmarksTable.cp`.

2. Examine the CBookmarksTable constructor.

`CBookmarksTable()` `CBookmarksTable.cp`

In the design of this application, the real work of setting up a table occurs in `InitBookmarksTable()`. You write that function in the next step. However, the process begins in the table constructor.

The CBookmarksTable constructor sets up a series of constants for cell height, indents for sublevels, text traits for the two kinds of items, and an icon ID for each kind of item. Once again, this code is provided for you. You don't have to type it in.

```
const Int16   kCellHeight = 16;
const Int16   kFirstIndent = 20;
const Int16   kLevelIndent = 16;
const ResIDT  kGroupTextTraits = 131;
const ResIDT  kBookmarkTextTraits = 132;
const ResIDT  kGroupIconID = 1001;
const ResIDT  kBookmarkIconID = 1000;
```

You could set up a CPPb resource in Constructor so that you could specify these values as part of the PPob data stream.

3. Create a table.

`InitBookmarksTable()` `CBookmarksTable.cp`

This is the step where you begin real work. As you know, to fully create a table you must attach helper objects, insert columns, insert rows, and install data.

a. Create helper objects.

The existing code in this function does some of the setup. It initializes the data members for indents, text traits, and icon IDs. It also creates two `STableItem` items: one group item and one bookmark item. These will be the default table entries.

After that, you can create the helper objects and store the pointers to the objects in the appropriate data members. There are three helper objects: the geometry, the selector, and the storage.

Use LTableMonoGeometry, and specify the width and height of a cell. Because this table will have a single column, the width of a

cell should be the width of the table frame, `mFrameSize.width`. The height is in the parameter `inCellHeight`.

Use LTableSingleSelector as the selection helper.

Use LTableArrayStorage as the storage helper. The size of the data is `sizeof(STableItem)`. Use the constructor to create an LArray for storage (as opposed to LVariableArray).

b. Insert a column.

The table has been created, and the helper objects have been attached to the table. Now you can insert columns and rows. Insert a single column in the table. Call `InsertCols()`. This column appears after column zero, there is only one column, with no data. There is no need to refresh the screen. The startup code will draw the window after creation.

c. Insert rows and install data.

So that there will be some default information, add two rows to the table. You have two items to add to the table, `theGroupItem` and `theBookmarkItem`.

Add the group item first as a sibling row. Call `InsertSiblingRows()`. You are adding one row, after row zero. Also pass the address of `theGroupItem`, the size of `STableItem`, a boolean value `true` (this is an expandable row), and a boolean value `false` (no need to refresh).

Then add the bookmark item as a child row of the group. Call `InsertChildRows()`. You are adding one row, as a child of row 1. Also pass the address of `theBookmarkItem`, the size of `STableItem`, a boolean value `false` (this is not an expandable row), and a boolean value `false` (no need to refresh).

The code for all three of these substeps is listed here. The remaining code adds an attachment that allows the user to use the page keys on an extended keyboard to scroll the view.

```
// Create helper objects for this table.
mTableGeometry = new LTableMonoGeometry( this,
                mFrameSize.width, inCellHeight );
mTableSelector = new LTableSingleSelector(this);
mTableStorage = new LTableArrayStorage( this,
                sizeof(STableItem) );
```

```
// Insert a single column.
InsertCols( 1, 0, nil, nil, false );

// Insert default items.
InsertSiblingRows( 1, 0, &theGroupItem,
            sizeof(STableItem), true, false );
InsertChildRows( 1, 1, &theBookmarkItem,
            sizeof(STableItem), false, false );
```

You have just created a table one column wide, two rows deep, with data in each cell. All cells are the same size, you can select one cell at a time, and the data storage for each cell is the same size.

4. Draw a cell.

`DrawCell()` `CBookmarksTable.cp`

For the table data to appear on screen, you must draw the contents of each cell. In this step you do the table-related setup work. The code that does the actual drawing is provided for you because it has no direct relevance to the table classes.

In a hierarchical table, any expandable item might be collapsed. As a result, a cell has two index values, one for a fully-expanded or wide open table, and one for its position among all the exposed cells. In this situation you need the wide open index.

To prepare for drawing, you need to do four things: get the wide open index, draw the expansion triangle, get the data for the cell, and get the nesting level of the cell.

a. Get the wide open index.

What you want is the index number for the row. The `DrawCell()` function receives the cell to be drawn through the `inCell` parameter. Call `GetWideOpenIndex()` and get the `inCell.row` index. This gives you the index for the row.

b. Draw the expansion triangle.

Call `DrawDropFlag()`. Pass `inCell` and the wide open index. PowerPlant takes care of the rest.

c. Get the cell data.

Call `GetItemFromCell()`. This is a custom function declared as part of the CBookmarksTable class. You write this function in the next step. The existing code declares an `STableItem` variable, `theItem`. This serves as the data buffer into which the cell's data will be placed. When you call `GetItemFromCell()`, pass `inCell` and `theItem`.

d. Get the nesting level.

Existing code later in this function uses the nesting level to determine how far to indent the cell's data so that child items appear indented under parent items. Existing code declares a `Uint32` variable, `theNestingLevel`. Call `GetNestingLevel()` to get that value. This is an LCollapsableTree function. You have a pointer to the tree helper object in `mCollapsableTree`.

The code for all substeps is listed here.

```
// Get the wide open index for the row.
TableIndexT theWideOpenIndex;
theWideOpenIndex = GetWideOpenIndex(inCell.row);

// Draw the cell drop flag.
DrawDropFlag( inCell, theWideOpenIndex );

// Get the cell data.
STableItem theItem;
if ( GetItemFromCell( inCell, theItem ) ) {

// Get the nesting level.
Uint32 theNestingLevel;
theNestingLevel = mCollapsableTree->
GetNestingLevel( theWideOpenIndex );
```

The remaining code in this function, provided for you, positions the icon (using the nesting level) for each item and draws the icon. It then draws the name of each item to the right of the icon.

5. Get data from a cell.

`GetItemFromCell()` `CBookmarksTable.cp`

In this step you complete `GetItemFromCell()` to retrieve the data from a particular cell. Existing code set `theDataSize` for an `STableItem`, and ensures that the data is valid.

To complete this function, you must do three things: get the wide open index for the row, create an `STableCell` object for the desired cell, and then get the data.

a. Get the wide open index for the row.

The value in `inCell` may contain the row and column for the exposed cell, as opposed to the wide open cell. You want the index value for the wide open row. Call `GetWideOpenIndex()`, pass `inCell.row`.

b. Create an STableCell for the cell.

Declare an `STableCell` variable. The solution code uses the name `theWideOpenCell`. Set its row to the wide open index value for the row. Set the column to 1.

c. Get the data from the cell.

Call `GetCellData()` for this wide open cell. The data buffer is the `outItem` parameter received by the call.

```
// Get the wide open index for the row.
TableIndexT theWideOpenIndex;
theWideOpenIndex = GetWideOpenIndex(inCell.row);

// Create an STableCell object for the cell.
STableCell theWideOpenCell(theWideOpenIndex, 1);

// Get the cell data (the bookmark item).
GetCellData(theWideOpenCell, &outItem,
            theDataSize);
```

`GetCellData()` copies the data from storage into the `outItem` buffer. All the data in storage is the size of an `STableItem`, and the buffer is declared in the caller to be an `STableItem`.

6. Insert an item in the table.

`InsertNewItem()` `CBookmarksTable.cp`

This is a complex step, because there are several situations that this function must handle. The item might be a group or a bookmark. Either item might be placed as a sibling or a child.

The algorithm implemented in this function decides how to place the new item based on whether there is a selected row. If the selected row is a group and the group is expanded, the new item is a child of the group. If the selected row is a bookmark or a collapsed group, the new item is a sibling of the selected row. If there is no selected row, the new item is placed at the top level of the hierarchy.

Existing code does three things.

It creates a new default item of the correct type, either a group or bookmark, and stores that in `theNewItem`.

It sets a boolean value `collapsable` to the correct value for the type of item. Groups can be expanded, bookmarks cannot. You'll use this value when you create a new row.

Existing code sets an `STableCell` variable, `theCell`, to (0,0). This variable ultimately holds either the selected cell, or the default value if no cell is selected. You start your search for a selected cell at (0,0).

a. Find a selected cell.

In the existing but empty `if` statement, call `GetNextSelectedCell()`. Pass `theCell` as the only parameter. The code you write in substeps **b** through **f** go inside this if statement and execute if a selected cell is found.

b. Get data from the selected cell.

Call `GetItemFromCell()` for the selected cell. Pass `theItem` as the data buffer. This variable is declared in existing code at the start of the function.

c. Get the wide open index for the row.

Call `GetWideOpenIndex()` for the selected cell's row.

d. Determine if the selected item is an open group.

There is an existing but empty `if` statement. Inside that `if` statement, perform two tests. First, look at the `mType` field of

`theItem`. This variable now holds the data retrieved from the selected cell. The desired type is `kGroupItemType`.

Also, determine if the group is expanded. Call the LCollapsableTree function `IsExpanded()`. Use the wide open row index.

If both tests pass, you execute the code in substep **e**. If either test fails, you execute the code in substep **f**.

e. Create a child item.

If both tests in substep **d** pass, then the selected item is an open group. Create a child row under that group. Call `InsertChildRows()` and pass in the appropriate parameters. The data is in `theNewItem`. The `collapsable` variable holds the appropriate boolean value for either group or bookmark items.

f. Create a sibling item.

If either test in substep **d** fails, then the selected item is either not a group or not open. Create a sibling row immediately after the selected row. Call `InsertSiblingRows()`. This code goes in the first existing `else` statement.

g. If no cell is selected, create an item at the start of the table.

All of the previous substeps related to a selected cell. If there is no selected cell, create a sibling row at the start of the table. This code goes inside the second existing `else` statement. Additional existing code inside this statement also sets `theCell.column` to the value 1 so the new cell can be properly selected in the next substep.

To create the sibling row, call `InsertSiblingRows()`.

h. Select the new cell.

After the `if/else` check for a selected cell, in all cases you want to select the new cell.

Call `UnselectAllCells()` to eliminate any existing selection.

Then, increment `theCell.row` by one, because the new row is one greater than the previous selected row. If there is no selected cell, `theCell.row` is zero, so incrementing makes the cell (1,1).)

Then call `SelectCell()`. You should check to ensure that the new cell is a valid cell by calling `IsValidCell()` before calling `SelectCell()`.

NOTE: The call to `UnselectAllCells()` is not really necessary if you use LTableSingleSelector. However, it is necessary for LTableMultiSelector.

The code for all these substeps is listed here.

```
STableCell theCell( 0, 0);

// If we find a selected cell.
if (GetNextSelectedCell (theCell)){

  // Get data from cell, assume it's valid.
  GetItemFromCell( theCell, theItem );

  // Get the row wide open index.
  theWideOpenIndex = GetWideOpenIndex(
                                  theCell.row);

  // If selected row is a group and is open.
  if (theItem.mType == kGroupItemType &&
      mCollapsableTree->IsExpanded(
                           theWideOpenIndex)){

    InsertChildRows( 1, theWideOpenIndex,
              &theNewItem, sizeof(STableItem),
              collapsable, true );
```

```
    } else { // not a group or not open

      // create sibling right after this one
      InsertSiblingRows( 1, theWideOpenIndex,
                &theNewItem, sizeof(STableItem),
                collapsable, true );
    }

  } else { // nothing selected

    // We enter here with theCell at (0,0).
    theCell.col = 1;

    // Add a sibling row at start of list
    InsertSiblingRows( 1, 0, &theNewItem,
        sizeof(STableItem), collapsable, true );
  }
  // Unselect all cells and select the new cell.
  UnselectAllCells();
  theCell.row++;
  if ( IsValidCell( theCell ) ) {
    SelectCell( theCell );
  }
```

7. Remove a row from the table.

`HandleKeyPress()` `CBookmarksTable.cp`

The last important task to accomplish is to remove a row from a table. In the example application interface, the selected row is removed when the user types the Delete key.

The existing code identifies the keypress, searches for a selected cell, and gets the wide open index for that row. You have already written code to perform the same tasks in previous steps.

After getting the wide open index, call `RemoveRows()`. Remember, with a hierarchical table you can only remove one row at a time. However, if that is a parent row, all children are removed as well.

```
theWideOpenIndex= GetWideOpenIndex(theCell.row);
RemoveRows( 1, theWideOpenIndex, true );
```

8. Examine other table features.

`various functions` `CBookmarksTable.cp`

You have performed all the principal tasks associated with tables in PowerPlant. The code provided for you performs some additional tasks worthy of a brief look.

CBookmarksTable overrides both `HiliteCellActively()` and `HiliteCellInactively()` to exclude the expansion triangle area from the cell highlighting. This is a common feature of hierarchical tables.

The application enables or disables some items in the **Bookmarks** menu based on selection. CBookmarksTable overrides `SelectionChanged()` to update menus when the selection changes. There is nothing table-specific going on here, but this does point out the function you override if you need to respond to a changed selection.

CBookmarksTable also overrides `ResizeFrameBy()`. This is an LView function. Because this table has a single column, that column fills the width of the frame. When the user resizes the window (and the table view contained therein), this function sets the column width to match the frame width. It uses `SetColWidth()`.

Finally, take a peek at `ClickCell()`. CBookmarksTable overrides this function so that when the user double-clicks a bookmark cell, the `OpenLocation()` function is called. The `OpenLocation()` function just beeps. However, this gives you a hook to implement code that would go out across the network and connect to the URL stored in that bookmark. Cool!

Finally, the code to edit the contents of a table cell has been completely provided for you. Most of that code concerns managing dialogs in which you edit the contents of the table cell. You have already mastered any table-related calls used in that code.

9. Build and run the application.

You're all done! When the application builds successfully and runs, the window shown in Figure 5.5 appears with one group, "Cool Sites," and a bookmark for the Metrowerks web page.

Use the **Bookmarks** menu to add new rows (items) to the table. Experiment with all the possibilities.

With no item selected, create a new group. It appears selected and open (with no contents yet) as the first item in the window. Add a new bookmark. It appears selected as a child of the new group. Create another new group. It appears as sibling to the bookmark you just made.

Click the expansion triangle to collapse a group. Select that closed group, and make a new group. It appears as a sibling to the selected group.

Continue to experiment with adding and removing table items. To remove an item, select the item and type the Delete key. If it is a parent row, all the children disappear.

When you are through playing with the demo, quit the application. There is plenty of room for further exploration on your way to mastery of the PowerPlant table classes.

For example, when there is no item selected a new row appears at the beginning of the table. Make the item appear at the end of table.

Change the currently selected cell in response to arrow keys.

Use LTableMultiSelector instead of LTableSingleSelector to manage cell selection in the table. Simply attach a different helper object. However, if you want to manage multiple selections, such as deleting multiple selected rows properly, you'll have to write some additional code.

Use LTableMultiGeometry so you can make the group rows taller than item rows. You can also design an interface to allow the user to set the height of selected rows.

Add a second column of data. Instead of resizing the table cell to match the width of the table frame, let the table become wider than the window. The horizontal scroll bar should then activate, so you can scroll left and right as well as up and down the table.

As always, have a good time exploring. PowerPlant's table classes can give you a real head start when you need to display tabular data.

6

Apple Events in PowerPlant

This chapter discusses how to work with Apple events in PowerPlant. The focus of this chapter is on how to create a scriptable application using Apple events. In the process, PowerPlant's Apple event classes are explained thoroughly.

Introduction to Apple Events in PowerPlant

As an experienced PowerPlant programmer, you know how easily PowerPlant handles the visual interface of a Macintosh application. The visual interface dispatches user interface events to operations on your application's data.

However, every System 7-savvy application should also have a second interface—an Apple event interface. The Apple event interface dispatches messages to the same set of operations as the visual interface, but without direct user action. The events come from a script or another application.

Apple events allow your users to automate lengthy tasks, exchange data between applications, or perform custom configurations. Apple events and the related AppleScript programming language are perhaps the most valued breakthrough of System 7, and are even more important in Mac OS 8.

PowerPlant makes coding the Apple event interface as easy as coding the visual interface. In fact, any PowerPlant application is already partially scriptable—PowerPlant's application, window, and document classes already support some Apple events. In this chapter, you will learn how to use PowerPlant classes to extend these built-in features to include your specific application content.

The discussion assumes that you are familiar with the purpose of Apple events, and all the basic AppleEvent Manager data structures: AEDesc, AERecord, and AppleEvent. You should also be familiar with the core suite of the Apple event registry (e.g. make new, clone, move, delete, get data, set data). Experience with scripting an application that supports the AppleEvent Object Model (such as BBEdit, CodeWarrior, FileMaker Pro, or Eudora) will speed your comprehension.

Where to Learn More About Apple Events

Apple events are a very big topic, occupying most of *Inside Macintosh: Interapplication Communications*. This chapter cannot possibly cover all the details of how Apple events work. This chapter is limited to those aspects of PowerPlant that you will need to use immediately to support Apple events in your application.

To learn more about Apple events, consult the following references.

Apple Computer, Inc. *Inside Macintosh: InterApplication Communication*. Addison-Wesley (1993).

Apple Computer, Inc.*Apple Event Registry: Standard Suite*. This document is on the Developer Reference CD (1992).

Berdahl, E. M. “Better Apple Event Coding Through Objects.” *develop*, 12, 58-83 (1992).

Clark, R. “Apple Event Objects and You.” *develop*, 19, 8-32 (1992).

Roschelle, J. “Powering Up AppleEvents in PowerPlant.” MacTech Magazine, 11(6), 33-46 (1995).

Simone, C. “According to Script: Steps to Scriptability.” *develop*, 24, 27-29 (1995).

Apple Event Strategy

Let's begin with a bird's eye view. Apple events are basically a way of describing the actions that users will perform in your application, such as:

- set a word's font to Helvetica
- make a graphic line thicker
- enter a formula in a spreadsheet

An Apple event is a purely descriptive message. It says what to do, not how to do it. Your application must parse and interpret this message. When you are finishing parsing and interpreting, you will call a normal C++ function to execute the operation. This suggests the main programming problem—translating an external message to internal classes and functions. Your Apple events code translates "what" into "how" and then executes the appropriate code.

To make this problem of translation easier, Apple events present messages in a fairly standard format. An event message has a verb that describes the operation to perform, and a direct object that describes the noun on which to perform the operation. The event may also have a number of parameters which, like adjectives and adverbs, specify how the operation is to be performed. For example, "set" (verb) "color of word 1" (noun) to "red" (adjective).

The Mac's visual interface is insanely great because Apple adopted strong user interface guidelines. Similarly, the Mac's Apple event interface is insanely great because Apple provided strong semantic guidelines for the Apple event interface. These guidelines are called the Apple Event Object Model (AEOM). AEOM is a generic vocabulary for modeling any application. You customize it to describe your application.The concepts in the AEOM vocabulary are:

- Classes—kinds of objects (nouns) in your application
- Events—actions (verbs) that can be performed
- Parameters—attributes of an action (adverbs)
- Properties—attributes of objects that can take on a value

- Elements—the hierarchical relationship between a class and the items it contains

NOTE: In the context of Apple events and the AEOM, the word "parameter" has a specific meaning. A parameter is one kind of data attached to an Apple event. Unfortunately, the word "parameter" also means data passed to a function. To avoid confusion, we will use the term "argument" when we are discussing data passed to a function, and the word "parameter" when discussing attributes of an Apple event.

These abstractions become concrete with the specifics of each application. For example, a spreadsheet document (class) contains elements which are cells (another class) and a cell has a formula, value, color, and border (properties). Similarly a graphics document (class) can contain rectangles (class) which have a fill color and line width (properties). You might sort (event) the cells in ascending order (parameter). Or move (event) the rectangle to the back (parameter).

The central task with Apple events is translating these messages into actions you can execute. In PowerPlant, the LModelObject class is the center of translation. The translation proceeds roughly like this:

- AEOM class—C++ class derived from LModelObject
- event—C++ member function of the class
- parameter—C++ argument to a member function
- property—C++ data member in the class
- elements—LList of contained LModelObjects in the class

Thus the message "set the line width of rectangle 3 of window 1 to 2" might be translated to C++ as:

```
theRectangle->SetLineWidth(2);
```

Line width might be stored in the member variable `mLineWidth`, and rectangle 3 could be the third pane in the `mSubPanes` of the first LWindow.

NOTE: Once again, there is potential confusion because of terminology. In the AEOM, the word “class” is a generic term referring to an object in your application. In C++, the word “class” means a formal description of a C++ object. There is usually a 1:1 relationship between these two kinds of classes. For every AEOM class, there is usually a corresponding C++ class that (in PowerPlant) inherits from LModelObject.

Apple Event Classes

In your Apple-event-savvy PowerPlant application, the center of attention is a mix-in class called LModelObject. LModelObject lets you “model” the data in your application as a tree of objects. You inherit from LModelObject in your main content-specific classes, and you override functions to respond to AppleEvents for each class.

The dirty work of translation between Apple event messages and C++ functions involves some helper classes in addition to LModelObject. The classes discussed in this section include:

- LModelObject—the principle PowerPlant class for implementing Apple events
- LModelDirector—wrapper for the low-level Apple event interface
- LModelProperty—helps LModelObject handle properties
- UExtractFromAEDesc—contains routines for decoding Apple events to C++ data types
- StAEDescriptor—manages AEDesc structures
- UAEDesc—encodes more complicated types of Apple event descriptors
- UAppleEventsMgr—contains routines for encoding C++ data types to Apple events

LModelDirector or LModelProperty are internal to PowerPlant. You will not usually use either class directly. However, you will use the classes and functions in UAppleEventsMgr and UExtractFromAEDesc because you will need to encode and decode Apple events.

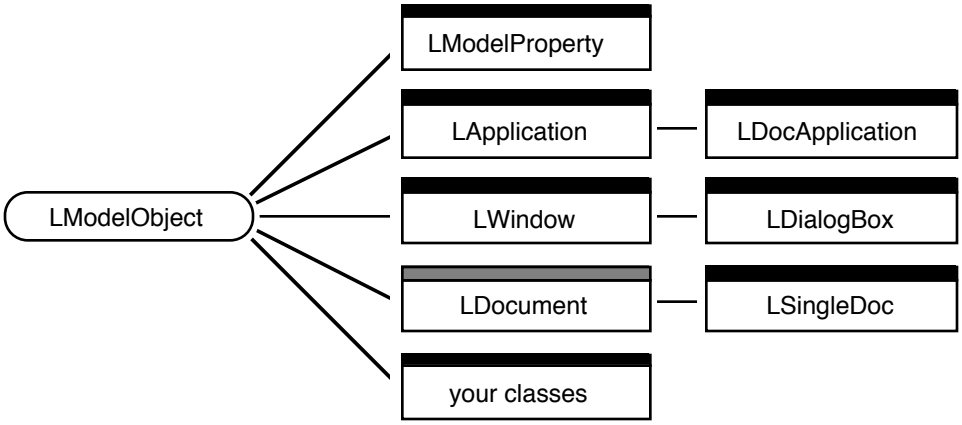
If you find yourself encoding or decoding large or complex Apple events, you may find it worthwhile to study the classes in UAEGizmos. UAEGizmos has faster routines for encoding and decoding AEDescs. These classes are not discussed in detail in this chapter, because they are unsupported. See "UAEGizmos" on page 193.

Finally, to use Apple events, you will also have to edit two resources: the terminology resource and the dispatch table. The terminology resource (of type 'aete') has two purposes. First, it describes your application's particular AEOM vocabulary. Second, it specifies the mapping between English-like terminology and codes that your C++ member functions can use. The other resource is the 'aedt' resource, which declares the numeric codes you use to represent Apple events.

LModelObject

A model object is a C++ class that handles a particular AEOM class in your application's Apple event interface. In PowerPlant, all model objects inherit from LModelObject. You inherit from LModelObject in the content-related classes in your application, which might be graphics shapes, spreadsheet cells, or word processing paragraphs. You will also find LModelObject already mixed into LWindow and LDocument (Figure 6.1). This is because windows and documents are part of the core suite in the Apple event registry, and PowerPlant does most of the work of supporting that suite for you.

Figure 6.1 Classes that inherit from LModelObject



The grey bar indicates that LDocument is an abstract class.

NOTE: You can refer to LWindow and LDocument as excellent examples of Apple event handling in PowerPlant. Beware, however, that both classes handle the elements relationship (finding a particular window or document) idiosyncratically. This is because lists of windows are stored in the Mac OS and the list of documents can be stored globally. Most of your classes will have to support the elements relationship differently.

LModelObject has three data members of interest.

Table 6.1 Some LModelObject data members

Data member	Stores
`mModelKind`	the type of the AEOM class of this object
`mSuperModel`	this object's containing LModelObject
`mSubModels`	an optional LList of contained elements

The `mModelKind` member is a 4-byte code that describes the AEOM class ID of an object. You typically set this member only when you first create an object. In advanced situations, you can change the member to allow one C++ class to support many AEOM classes. The next two members, `mSuperModel` and `mSubModels`, support the AEOM elements relationship. Since every LModelObject can have a list of other LModelObjects, you can establish a model object hierarchy. At any level in the hierarchy, the `mSuperModel` member refers to the immediate container of the given LModelObject, and the `mSubModels` member is an LList of its elements.

NOTE: The `mSubModels` LList is optional. To use it, call `SetUseSubModelList(true)` in your object's constructor. You will not use `mSubModels` if your object contains no elements or you implement custom storage of elements. For example, an application might store the elements in a hash table. In this case `mSubModels` would be nil. You would have to override some of the member functions of LModelObject such as `AddSubModel()` and `GetSubModelBy()`.

LModelObject has more than 50 member functions. Many are internal to PowerPlant's handling of Apple events. The member functions that you are likely to call or override fall into three categories:

- Managing elements—the contents of an object
- Handling properties—the attributes of an object
- Handling events—the actions performed on objects

Managing elements

The AEOM specifies a containment hierarchy according to the elements relationship. LModelObject provides several functions for managing this relationship.

Table 6.2 Basic LModelObject functions for managing elements

Function	Purpose
`constructor`	sets model kind and location in hierarchy
`SetUseSubModel-List()`	initializes `mSubModels` member
`SetModelKind()`	changes an LModelObject's AEOM class ID
`GetModelKind()`	returns an LModelObject's AEOM class ID
`SetSuperModel()`	changes an LModelObject's container
`GetSuperModel()`	returns an LModelObject's containing LModelObject.
`IsSubModelOf()`	tests this object for membership in container
`GetPositionOf-SubModel()`	returns the index of this object in its container

In the recommended LModelObject constructor, you provide both a pointer to a containing LModelObject and a class ID. In one step, you create an object of a particular AEOM class in a particular containing model. Alternatively, you can create an LModelObject with the default constructor, and later set the AEOM class ID with `SetModelKind()`. You can change the containment at any time by calling `SetSuperModel()` with a new containing LModelObject.

If your object will have elements, you might want to use the `mSubModels` member to store them. If so, your constructor implementation should also call `SetUseSubModelList(true)`.

NOTE: When you add an element, you cannot directly specify its position in the list of elements. After you call `SetSuperModel()`,

you can change the position using the `MoveItem()` function of the `mSubModels` LList.

Table 6.3 Advanced LModelObject functions for managing elements

Function	Purpose
`AddSubModel()`	adds an element to this container
`RemoveSubModel()`	removes an element from this container
`GetSubModelBy-Position()`	returns an element by an integer index
`GetSubModelByName()`	returns an element by its name
`GetPositionOfSubMod-el()`	returns the index of an element
`CountSubModels()`	returns the number of submodels

You might not want to use the `mSubModels` list to store your elements. For example, you might have a very large collection of elements that are accessed by name. In this case, storing your elements in a hash table would provide more efficient lookup. However, if you do not use the `mSubModels` list, you have to override the member functions listed above so that PowerPlant can find your elements.

The `AddSubModel()` and `RemoveSubModel()` functions are called by `SetSuperModel()` and by the recommended constructor. Your overrides should insert or delete the specified LModelObject in your data structure.

`CountSubModels()` should return the number of elements of the desired class of your data structure. It is called when an Apple event asks for information like the "number of rectangles in window 1." This function is also called when an Apple event requests the "last rectangle of window." PowerPlant counts the number in the list and translates the request into a request for an indexed item.

The `GetSubModelByPosition()` and `GetSubModelByName()` member functions should find the desired element in your data structure. They are called when an Apple event requests something like "rectangle 4 of window 1" or "rectangle "fred" of window 1." `GetPositionOfSubModel()` should return the index number of an element in your storage.

In addition to these functions, LModelObject provides many more functions you can override to support more complex accessors for your elements. You can allow users to name an object by a unique identity or by a "whose" clause. You can support comparisons between objects. You can find the appropriate functions to override in `LModelObject.h`.

Handling properties

Properties are the attributes of an object, such as color, line width, location, size, font, and so forth. LModelObject has functions that are, effectively, accessors for properties.

Table 6.4 LModelObject functions for handling properties

Function	Purpose
`GetAEProperty()`	returns a property value to an Apple event
`SetAEProperty()`	sets a property value from data in an Apple event

To support any properties in your LModelObject-based classes, you override these two functions. Note that in both functions, the first argument is the property descriptor. Typically, you store each property in a data member of your class. Your implementation of both `GetAEProperty()` and `SetAEProperty()` will contain a switch statement that maps the property descriptor to a particular data member.

NOTE: PowerPlant translates the "Get Data" and "Set Data" Apple events into calls to `GetAEProperty()` and `SetAEProperty()` with a property ID of `pContents`. Therefore an Apple event

like “set rectangle 4 to {10,0,50,100} ” results in a call to `SetAEProperty()`.

Handling events

PowerPlant handles Apple events through a central dispatcher. When an event is received, PowerPlant first identifies the LModelObject (the direct object) to which the event applies. The `HandleAppleEvent()` member function of this object will be called. `HandleAppleEvent()` decodes the event ID and dispatches to an appropriate function to further decode and execute the event. PowerPlant already dispatches the following Apple events:

- make new (create element)
- clone
- move
- delete
- get data
- get data size
- set data
- count elements

If you support other events, you must override `HandleAppleEvent()` to dispatch to your functions.

Table 6.5 LModelObject functions for handling events

Function	Purpose
`HandleAppleEvent()`	dispatches Apple events
`HandleCreateElementEvent()`	override to create a new LModelObject
`GetImportantAEProperties()`	gets the data to be cloned
`HandleMove()`	moves an element

Function	Purpose
`HandleClone()`	clones an element
`HandleDelete()`	deletes an element
`HandleCount()`	counts elements in a container

If you allow users to create new elements by an Apple event, you will have to override `HandleCreateElementEvent()`. Your implementation should create an LModelObject of the appropriate class and insert it in the right spot in its container. (Implementation details for `HandleCreateElementEvent()` and other member functions are discussed in "Implementing Apple Events in PowerPlant" on page 188.

`GetImportantAEProperties()` is called by `HandleClone()`. `HandleClone()` works by translating the clone request into a "Create Element" Apple event. To replicate the properties of the target object, it calls `GetImportantAEProperties()` and puts them into the properties field of the "Create Element" Apple event. If you want cloning to replicate your properties, you must implement `GetImportantAEProperties()` to build a Apple event record containing all properties of your object.

`HandleMove()` works by cloning the object into the new location, and deleting it from the old location. As mentioned above, cloning works by creating a new element. `HandleDelete()` arranges for the delete operator to be applied to your object. The LModelObject destructor will remove your object from its container. `HandleCount()` calls `CountSubModels()`.

In most cases you do not need to override `HandleMove()`, `HandleClone()`, `HandleDelete()`, or `HandleCount()`. You would only override these member functions if you can provide more efficient implementations.

LModelDirector

LModelDirector is used internally by PowerPlant. You will usually not need to call any LModelDirector member functions yourself, although `LModelDiretor::Resolve()` may be useful.

In a running PowerPlant application, there is one instance of LModelDirector, created at launch time. This instance installs callback handlers into the Toolbox Apple Event Manager. When your application receives an event, LModelDirector performs the initial decoding of the event, and dispatches to an appropriate LModelObject class to be handled.

LModelProperty

LModelProperty is used internally by PowerPlant to handle properties of classes. You will typically not need to make LModelProperty instances yourself, nor call LModelProperty member functions.

PowerPlant transiently creates LModelProperty instances as they are needed, to represent an AEOM property in an Apple event. A property is always contained by an LModelObject instance. LModelProperty dispatches back to this instance to execute its set or get property Apple events. Thus, you always handle property-related Apple events in the `GetAEProperty()` and `SetAEProperty()` member functions of your LModelObject-based class.

UExtractFromAEDesc

The UExtractFromAEDesc class decodes Apple event descriptors to C++ data types. It consists entirely of static functions, one per data type.

NOTE: UAEGizmos has this functionality as well, in the `LAESubDesc::To...()` functions.

Table 6.6 UExtractFromAEDesc functions

Function	Purpose
`TheInt32()`	decodes an Int32
`TheInt16()`	decodes an Int16
`ThePoint()`	decodes a QuickDraw Point
`TheRect()`	decodes a QuickDraw Rect
`TheBoolean()`	decodes a boolean
`TheType()`	decodes a class ID (DescType)
`TheEnum()`	decodes an enumeration constant (DescType)
`TheRGBColor()`	decodes a QuickDraw color
`ThePString()`	decodes an Str255

For example, if you want to extract a long integer from an Apple event descriptor named `inDesc`, you can call:

```
Int32 myInteger;
UExtractFromAEDesc::TheInt32(inDesc, myInteger);
```

The virtue in using these functions to decode data from an Apple event is that they will automatically coerce Apple event data into the desired type. Thus if your application wants an Int32, but the Apple event supplied an Int16, your code will still get data. (Incidentally, it is because of coercion that `TheType()` and `TheEnum()` are both provided. Even though they produce the same C++ data type, they might perform different coercions.)

If coercion fails, these member functions throw an exception. In most cases, this does the right thing for you automatically. The exception will get caught by PowerPlant code in LModelDirector and translated into an error code. The error code will be returned in the Apple event reply. From there, the sending application can deal

with the error gracefully. Script Editor, for example, will tell the user in which line of AppleScript the error occurred.

StAEDescriptor

Whereas UExtractFromAEDesc gets a value out of a descriptor, StAEDescriptor gets a descriptor corresponding to a particular parameter out of an Apple event. You can also use StAEDescriptor for constructing Apple event descriptors that are needed temporarily.

StAEDescriptor (in `UAppleEventMgr.h`) wraps an Apple event descriptor with a stack-based C++ class. When the local block of code completes, StAEDescriptor will be destructed and will properly dispose of the Apple event descriptor. This is necessary because the AppleEvent Manager copies the parameter from the Apple event into an Apple event descriptor. Programmers are responsible for disposing of this new descriptor. StAEDescriptor does this for you automatically. Because it is a stack-based class, it works even when an exception is thrown.

Table 6.7 Some StAEDescriptor functions

Function	Purpose
`GetParamDesc()`	retrieves an Apple event parameter
`GetOptional-ParamDesc()`	as above, but doesn't throw an error if not found

Every parameter in an Apple event has a keyword that identifies it. StAEDescriptor finds a descriptor by keyword and makes it available for your use. You can then use a UExtractFromAEDesc function to decode the value of the parameter. For example, if you received an Apple event that had an "index" parameter containing a long integer, you could retrieve the data as follows:

```
StAEDescriptor indexDesc;
Int32 mylong;
indexDesc.GetParamDesc(theAppleEvent,keyIndex,
```

```
                        typeLongInteger);
UExtractFromAEDesc::TheInt32(indexDesc,myLong);
```

You should use `GetParamDesc()` for a parameter that is required, and `GetOptionalParamDesc()` for a parameter that is optional. The only difference is that the former member function will throw an exception if the required parameter is not found. Under normal PowerPlant handling, this exception will cause the Apple event to return an error to its caller.

As a safety check, after you retrieve all the required and optional parameters of an event you should call `UAppleEventMgr::CheckForMissingParameters()`. This function throws an exception if you forgot to retrieve a parameter from the message.

The StAEDescriptor class also has a second usage. You can use it to encode C++ data while building an Apple event. StAEDescriptor has an overloaded set of constructors, each of which encodes a descriptor from a different class of C++ data type. Moreover, because StAEDescriptor defines some cast operators, you can use an StAEDescriptor anywhere that the AppleEvent Manager needs an Apple event descriptor. For example, to add the long integer `myLong` to an Apple event you could use code like this:

```
StAEDescriptor aeLong(myLong);
::AEPutParamDesc(myAppleEvent,keyIndex,aeLong);
```

This conveniently first constructs the Apple event encoding of the data in `myLong`, and then disposes the Apple event descriptor for `aeLong` after it is added to the event.

WARNING! Classes that have two, incompatible uses are dangerous. Unfortunately StAEDescriptor is just such a class. If you are decoding an Apple event, you must use the no-argument constructor and call either of the functions listed above. If you are encoding an Apple event, you generally use the constructor that takes an argument, and do not call other StAEDescriptor functions.

UAEDesc

The UAEDesc (in `UAppleEventMgr.h`) class encodes more complicated types of Apple event descriptors. The static member functions in this class simplify the process of encoding lists and records.

Table 6.8 UAEDesc functions

Function	Purpose
`AddPtr()`	adds a C++ variable to a list
`AddDesc()`	adds an Apple event descriptor to a list
`AddKeyDesc()`	adds an Apple event descriptor to a record
`MakeRange()`	encodes a range descriptor
`MakeInsertionLoc()`	encodes an insertion location
`MakeBooleanDesc()`	encodes a boolean

The three "Add" functions will create a list or record if necessary, and then add an item. For example, to create a list of the integers from 1 to 10 you could write:

```
StAEDescriptor myList;
for( long i = 1; i <= 10; ++i)
{
  UAEDesc::AddPtr(myList,i,typeShortInteger,
                  &i, sizeof( short ));
}
```

`MakeRange()` is used to encode an object descriptor that denotes a set of objects. An example is "characters 1 through 5 in word 1." `MakeInsertionLoc()` encodes a description of a place where an object should be created, cloned, or moved.

TIP: Once again, refer to the UAEGizmos package if you intend to work with complex descriptors.

UAppleEventsMgr

The UAppleEventsMgr class contains functions for sending an Apple event to your own application, as well as some other utilities.

Table 6.9 Some UAppleEventsMgr functions

Function	Purpose
`MakeAppleEvent()`	creates an event
`SendAppleEvent()`	sends an event
`SendAppleEventWithReply()`	sends an event and receives a reply

The general procedure for sending an Apple event is to:

1. Make a descriptor for the Apple event using StAEDescriptor.
2. Call `MakeAppleEvent()` to encode the event class and ID into the descriptor.
3. Add parameters to the descriptor using `AEPutParamPtr()` or `AEPutParamDesc()`. (You usually want to build the parameters with StAEDescriptor and UAEDesc.)
4. Send the Apple event.
5. If appropriate, decode the reply.

WARNING! `SendAppleEvent()` and `SendAppleEventWithReply()` are inconsistent. The former disposes the Apple event for you, the latter does not. The best policy is to create the AEDesc for your Apple event and its reply using StAEDescriptor. This will ensure that the descriptors are deallocated exactly once.

Apple Event Resources

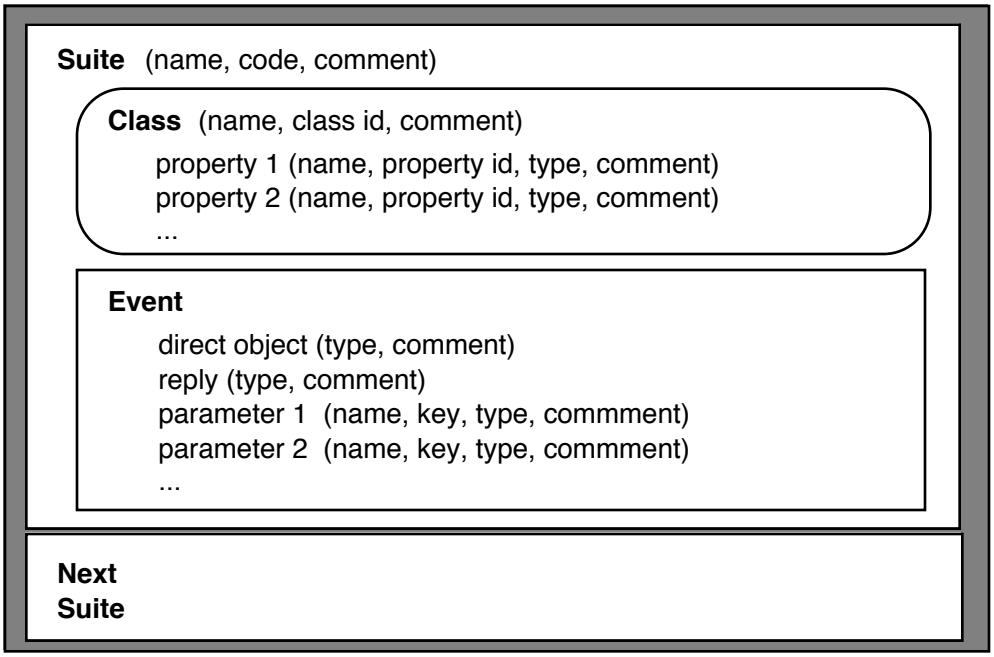
In order to build a scriptable application, you must modify two resources. This section has the following topics:

- The ‘aete’ Resource—Apple event terminology extension
- The ‘aedt’ Resource—Apple event dispatch table
- Editing Apple Event Resources—starter files and Resorcerer

The ‘aete’ Resource

The terminology resource describes the AEOM classes of objects and events that your application supports. It also enables AppleScript to translate English-like verbs and nouns into four-letter codes that your application can easily process. The English-like terms are called “user terminology” or “user terms” for short. The four-letter codes are of type `DescType`, and are stored in a single long integer. Figure 6.2 shows the features of an ‘aete’ resource.

Figure 6.2 Terminology resource



The terminology is organized by suites, with each suite covering a standard kind of functionality. The standard suites are described in the AppleEvent Registry, available on the Apple developer program Reference CD. Standard suites cover text, graphics, and tables, for example. You are free to invent your own suites, but if a standard suite fits your application you should strive to use it.

Within each suite, you describe a set of events and a set of classes.

Each event description gives the user terminology and the event ID for the event, as well as a descriptive comment. Within the event is a list of parameters, again described both with a user term and a code.

Each class description gives the user term, class ID code, and descriptive comment for the class. Within the class description, there is a set of properties. Each property has both a user term and a code, as well as a description.

A class can also have a set of elements. These are described simply by listing the class IDs of the objects that can be contained within this class.

Here are a few tips for editing terminology resources:

1. Mappings from user term to application codes must be one to one. Never have two user terms for the same code or vice versa, even if they are in different suites.
2. AppleScript does not support static type checking of a script (in contrast to C or Pascal). Thus upon checking syntax, AppleScript reports correct syntax for any property with any class, and for any element in any class. Even though your terminology describes a data type for each property and parameter, AppleScript lets the user send a different data type. AppleScript has no way to indicate that certain events only apply to certain classes. Because AppleScript is weakly type checked, your application may report errors at run-time even thought the script was “correct” at compile time.
3. AppleScript caches terminology resources. If you change your terminology, you must quit and re-launch Script Editor or its equivalent if you want to use the new terminology.

The 'aedt' Resource

The 'aedt' resource simply maps two pieces of information to one. The event class and event ID are mapped to a single long integer code. You use the long integer code inside your implementation. If you add any events to any suite, you must supply an entry in an 'aedt' resource, or PowerPlant will not dispatch your event.

Editing Apple Event Resources

The file `<PP Starter Resource>.rsrc` includes both "aete" and "aedt" resources for basic PowerPlant operations. You can use these resources as a starting point for your own scriptable application. You can find these files in the Project Stationery support folder.

The starter 'aete' resource can also be found in the file `PP Copy & Customize.rsrc`. The starter 'aedt' resource can also be found in the file `PP AppleEvents.rsrc`. See the "Resource Notes" chapter of *The PowerPlant Book* for more information on these files.

To edit an 'aete,' you can use ResEdit, Rez, or the Resorcerer resource editor. The instructions in this chapter youse Resorcerer. A ResEdit template for aete editing is currently available at:

```
ftp://ftpdev.info.apple.com/Developer_Services/
Tool_Chest/Interapplication_Communication/
AE_Tools_/ResEdit_'aete'_Editor_1.0b4.sit.hqx
```

For a Rez version of these resources, see the file `PPSuites.r` and its siblings.

Implementing Apple Events in PowerPlant

Apple event implementations can vary widely in sophistication. Beginners should start with the basics to avoid being overwhelmed. The basic steps for any implementation are:

- Adding Classes—add AEOM classes (and their elements) to the AEOM hierarchy for your application. Support `HandleCreateElementEvent()` so users can create new elements.

The other core events (e.g. clone, move, delete) will be handled for you by PowerPlant.

- Adding Properties—so that users can get and set the state of each object in your application.
- Adding Custom Apple Events—add any needed events to `HandleAppleEvent()`, so users can perform application-specific actions with your objects.

After mastering these steps, you may want to read:

- Beyond the Basics—a peek at more advanced Apple event techniques

Adding Classes

The standard PowerPlant terminology resource includes classes for the application, and its documents and windows. You will probably want to add additional classes that describe the objects within your documents and windows.

The first step involves editing the terminology resource.

1. Add an entry for each new class and assign it an appropriate code. Edit the containing classes (e.g. window) so they list your classes as elements.

In the next two steps you modify your C++ objects that implement each AEOM class.

2. Add LModelObject as a public ancestor of your C++ class.
3. Redefine the constructor so that it takes an LModelObject reference to its container. Pass this pointer to the LModelObject constructor.

In the third step you add your object to its container when it is constructed. Obviously, you have to change each place you call the constructor too, so you pass in an appropriate container.

In the last two steps, you also have to modify the container.

4. In each container (e.g. a window), call `SetUseSubModelList(true)` in the constructor to activate PowerPlant's default mechanism for handling contained elements.

5. In each container, implement `HandleCreateElement-Event()`.

A typical implementation of `HandleCreateElementEvent()` uses a switch statement to translate between a class ID and `operator new`.

Listing 6.1 Typical `HandleCreateElementEvent()` code

```
LModelObject* YourContainer::
                        HandleCreateElementEvent(
  DescType inElemClass,
  DescType inInsertPosition,
  LModelObject* inTargetObject,
  const AppleEvent& inAppleEvent,
  AppleEvent& outAEReply)
{
  LModelObject *result = nil;
  switch(inElemClass) {
    case myClassID:
      result = new myClass(this);
      break;

    case myOtherClassID:
      result = new myOtherClass(this);
      break;

    default:
      throw(errAEEventNotHandled);
      break;
  }
  return result;
}
```

Note that the constructor for each object passes a reference to the container (i.e. `this`). Your constructor calls LModelObject's constructor with this reference, so the new object is added to the container.

After constructing the new object, you might want to adjust its position in the list. To do this, use the `inInsertPosition` and `inTargetObject` arguments to locate the desired location. Then move the object there with `MoveItem()`.

You might also want to parse the Apple event for the "with data" and "with properties" parameters. Use these to configure the new object. Finally you probably need to cause your object to display itself in the visual interface.

Adding Properties

To enable users to get and set the value of properties for your classes, follow these steps.

1. Edit the 'aete' resource, adding property names and codes for each of your AEOM classes. Make sure each property name has a consistent code within the entire terminology, and vice versa.
2. Override `GetAEProperty()` to encode your C++ data into an Apple event descriptor.
3. Override `SetAEProperty()` to set your C++ data to the result of decoding an Apple event descriptor.

Both functions typically have a switch statement that handles each possible property as a case. Remember that the property with code `pContents` should correspond to the central datum in your class. Examine the implementation of LWindow for examples of how to write these two functions.

Adding Custom Apple Events

You should be able to get some functionality working using only the core events that PowerPlant already dispatches. The next step is to support standard events. Standard events such as make, delete, copy, move, duplicate, get, and set are common and familiar to scripters.

Eventually, you may need to add an event that is specific to your application. As before, you start by editing resources.

1. Edit the 'aete' resource, adding the new event and its parameters to an appropriate suite.
2. Edit the 'aedt' resource, mapping the event class and ID into a unique long integer.
3. Override `HandleAppleEvent()` in those classes that support the event.

A typical implementation of `HandleAppleEvent()` uses a switch statement to dispatch on the long integer that represents your event. You typically call a handler function, and pass it the Apple event. The handler function should extract the required and optional parameters it needs from the event, and then execute the appropriate application-specific action. Examine `LModelObject::HandleAppleEvent()` as a typical implementation of this function.

NOTE: Remember to call the inherited `HandleAppleEvent()` function in the default case so that standard events will be handled for you.

Beyond the Basics

After you get classes, properties and events working, there is much you can do to improve your Apple event interface. PowerPlant has hooks for many other features. Some of those features are:

- Laziness
- Default submodels
- UAEGizmos
- Whose clauses
- Recordability

Laziness

The "laziness" feature of LModelObject can be used in cases where it would require too much space to create a C++ object for every

model object. Suppose you have a scientific plotting program. You might not be able to feasibly keep an individual C++ object for every data point (a large array for all the points would be more efficient). In this case, you can create an LModelObject transiently, as needed to interpret a particular Apple event. This type of model object is called a "lazy" object.

To implement a lazy object strategy, you override the object accessor functions like `GetSubModelByPosition()` to actually create a new LModelObject to represent the desired data point. After creating the object, call `SetLaziness(true)`. At the end of the execution of the Apple event, PowerPlant will automatically de-allocate the lazy object. Examine LModelProperty as an example of a lazy object.

Default submodels

The "default submodel" and "set tell target" features of LModelObject allow you to simplify your scripting vocabulary. You might find that it takes very long chains of references to identify a particular object. In some cases, you can reduce this effort by making a particular object a default submodel of a particular container. The script author can then leave out the reference to the default object, and the script will still work. `SetTellTarget()` allows you to modify how AppleScript records your application, enabling it to use "tell" directives to make the script more readable.

UAEGizmos

You may want to use UAEGizmos to encode and decode Apple event descriptors. UAEGizmos is faster and easier to use than UAEDesc. UAEGizmos provides C++ wrappers for and relies on the AEGizmos library. The AEGizmos library is not a Metrowerks product, and Metrowerks does not support either AEGizmos or the UAEGizmos classes. However, they are useful for Apple event programming. Read the AEGizmos documentation. You can find this material in the AEGizmos folder. The path to this folder is `PowerPlant:· In Progress:· AppleEvent Classes`.

You might also be able to implement a more efficient way of storing elements in containers. You could override LModelObject's functions that use `mSubModels`.

Whose clauses

Further down the line, you might want to support "whose" clauses. A "whose" clause allows a script author to refer to a whole collection of objects at once (e.g. every word whose font size is 12). To implement a whose clause strategy, you override `GetSubModelByComplexKey()`.

Recordability

You probably will also want to make your program recordable. In a recordable application, every user interface event sends an Apple event that represents the transaction. All changes to your data model thus occur through Apple events.

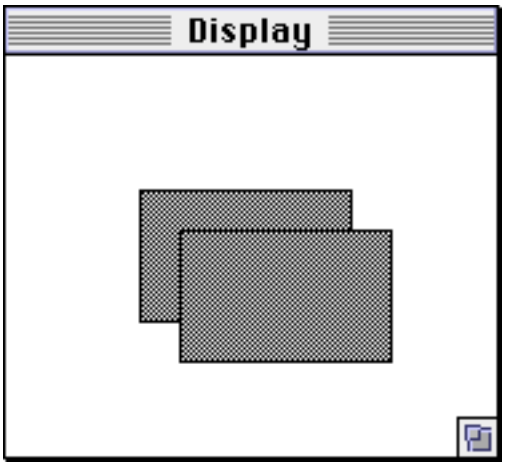
Recordability should be a primary goal right from the start. Design your application to be recordable. Users can learn how to script your application much more quickly by looking at the script recordings.

LWindow is recordable. You can look at how it translates user interface events like dragging the window position into Apple events. LModelObject also has a `SendSelfAE()` function to make it easier to send an Apple event to yourself.

Code Exercise for Apple Events

In this exercise you create a scriptable PowerPlant application that draws rectangles in windows and moves them around. To keep things simple, the application focuses entirely on scripting a freshly created window. There is no way to create or modify the objects in the window other than by scripts. The application's window is shown in Figure 6.3.

Figure 6.3 The Apple event test window



To make this exercise even easier, the application includes a **Script** menu. Scripts that demonstrate the scriptable features of the application are included for your use, and appear automatically on the **Script** menu. The scripts let you create, delete, rotate, change the line size, and change the fill type of a rectangle within the window.

As a bonus, the **Script** menu code is included in the sample project. You can attach any script to the application simply by placing the script in the Script Menu Items folder. The application scans this folder at launch time and adds each script file to the **Script** menu.

The required code is provided whole and complete, and requires no work on your part. Feel free to examine it, and use it in your own projects. The strategy used to implement the **Script** menu is the same used to implement a **Window** menu in Chapter 15 of *The PowerPlant Book*. The menu is an attachment to the application.

Being example code, there are a couple of things about the code that you may wish to avoid in your own projects.

First, this code assumes the presence of the AppleScriptLib. You might want to import weak for this library, check for AppleScript at runtime, and display a friendly alert if AppleScript is not present.

Second, this code relies on the use of PowerPlant precompiled headers. The prefix to include the precompiled header is set in the C/C++ Language preferences. As a result, the source files do not include many PowerPlant files that would otherwise be required.

Now that we have those little caveats out of the way, let's look at what you do in this exercise.

To implement Apple event support in this code exercise, you will edit the resources and write the code necessary to create the rectangle as an AEOM class. You also give each rectangle a property for its line width and fill. Finally, you add a custom event that rotates the rectangle by 90 degrees.

This exercise has four major sections. These sections mirror the basic tasks you must accomplish when creating a scriptable application. In this code exercise you:

- Edit Apple Event Resources—steps 1-5
- Create a Model Object in the Application—steps 6-10
- Add Model Properties to the Class—steps 11-12
- Add a Custom Event to the Application—steps 13-14

In addition, there is an optional section:

- Improve HandleCreateElementEvent()—steps 15-17

Let's get started.

Edit Apple Event Resources

You begin the process by editing the 'aete' and 'aedt' resources to expose the correct Apple event terminology for this application. The outline below describes what your edit should accomplish:

- Core Suite
  - Events—unchanged
  - Document class

    add shape element (step 1)
- Add shape suite (step 2)

  - add rotate event (step 2)
  - add shape class (step 3)
    - add line width property (step 4)
    - add filled property (step 4)
- Edit the 'aedt' resource for the rotate event (step 5)

Steps 1-5 require that you use Resorcerer, a commercial resource editor. If you do not own Resorcerer, you have three alternatives.

1. You can skip steps 1-5. Instead, copy the `AETest.rsrc` file from the solution code and replace the file of the same name in the start code. This file contains the project-specific 'aete' and 'aedt' resources you create in steps 1-5. You can get an 'aete' editor for ResEdit from
2. You can use ResEdit if you add an 'aete' editor to ResEdit. You can find such an editor at:

   ```
   ftp://ftpdev.info.apple.com/
   Developer_Services/Tool_Chest/
   Interapplication_Communication/AE_Tools_/
   ResEdit_%27aete%27_Editor_1.0b4.sit.hqx
   ```

3. You can derez the resource file, make the modifications in Rez, then rerez the file.

In steps 1-5, as you edit the 'aete' resource you specify the codes that apply to the new AEOM class, properties, and event. The file `AETestDef.h` defines constants for the codes you use for class, property and event IDs. If you use the values specified in the steps, you won't have to worry about changing the corresponding definitions in `AETestDef.h`, and everything should work fine.

1. Add a shape element to the window class.

`'aete' resource` `AETest.rsrc`

The window class already exists in the start code 'aete' resource. Open the 'aete' resource in Resorcerer, and then locate the window class. To simplify navigation, you may wish to turn on the **Show Index Popups** item in the Resorcerer **Custom** menu.

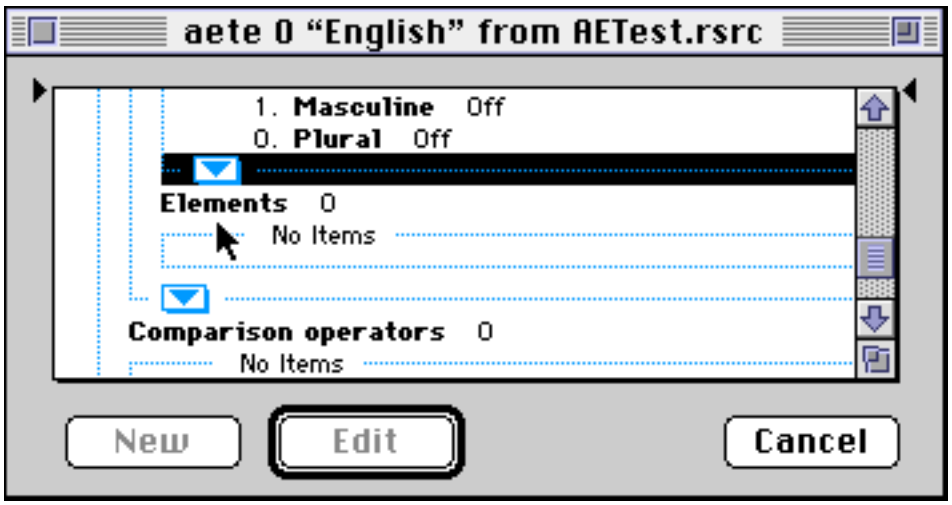
If you do, use the index popups to go to the second suite. Go to the end of that suite's events to see that suite's classes. Go to the second

class, the window class. Go to the end of the window class's properties. Elements are listed after properties. There are currently no elements allowed in this window. In this step you add an element.

When you locate the right spot, the Resorcerer window should look something like Figure 6.4. Be careful you've got the right spot. You want to add an element to the window class. This window will contain rectangles. The ID code for the AEOM shape class is 'cShp.'

To create a new element, drag the insertion point triangle (on the left side of the window) down to the **No Items** entry under the **Elements** category. Then click the **New** button. An **Element Class Code** appears with the default value of AEList. There is a popup menu with other predetermined options. You are creating a custom object, so the code for that object does not appear on the popup menu.

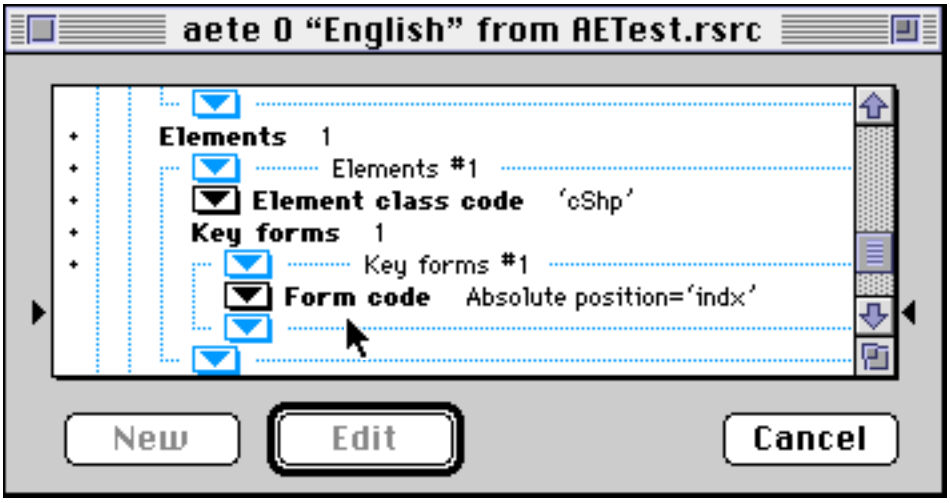
Figure 6.4 The Resorcerer 'aete' window before changes



Double click the **Element Class Code** item to edit its value. In the resulting dialog, set the class code to 'cShp' and close the dialog.

Under the **Element Class Code** item is a list of **Key forms** indicating there are none. Move the insertion point triangle to that location, and create a new key form. When you do, you get an entry for the **Form Code**. The default value is "Absolute position." This is just what you want. There is no need to change this value. When you are finished, the aete resource looks like Figure 6.5.

Figure 6.5 After completing step 1



You haven't actually created the 'cShp' class yet. You do that in Step 3. In this step you have said that objects of the 'cShp' class can go inside a window in this application.

2. Create a new suite and add a rotate event.

`'aete' resource` `AETest.rsrc`

Scroll to the very end of the 'aete' resource, and move the insertion point triangle to the very end of the file as well. Then click the **New** button to create a new suite. This is Suite #5.

You can edit the name of the suite. In the solution code, this is the Shape Suite.

By default, the suite code is 'reqd.' You must edit this value. You are adding a custom suite. Double click the **Suite code** item. In the resulting dialog set the code to 'sShp,' then close the dialog.

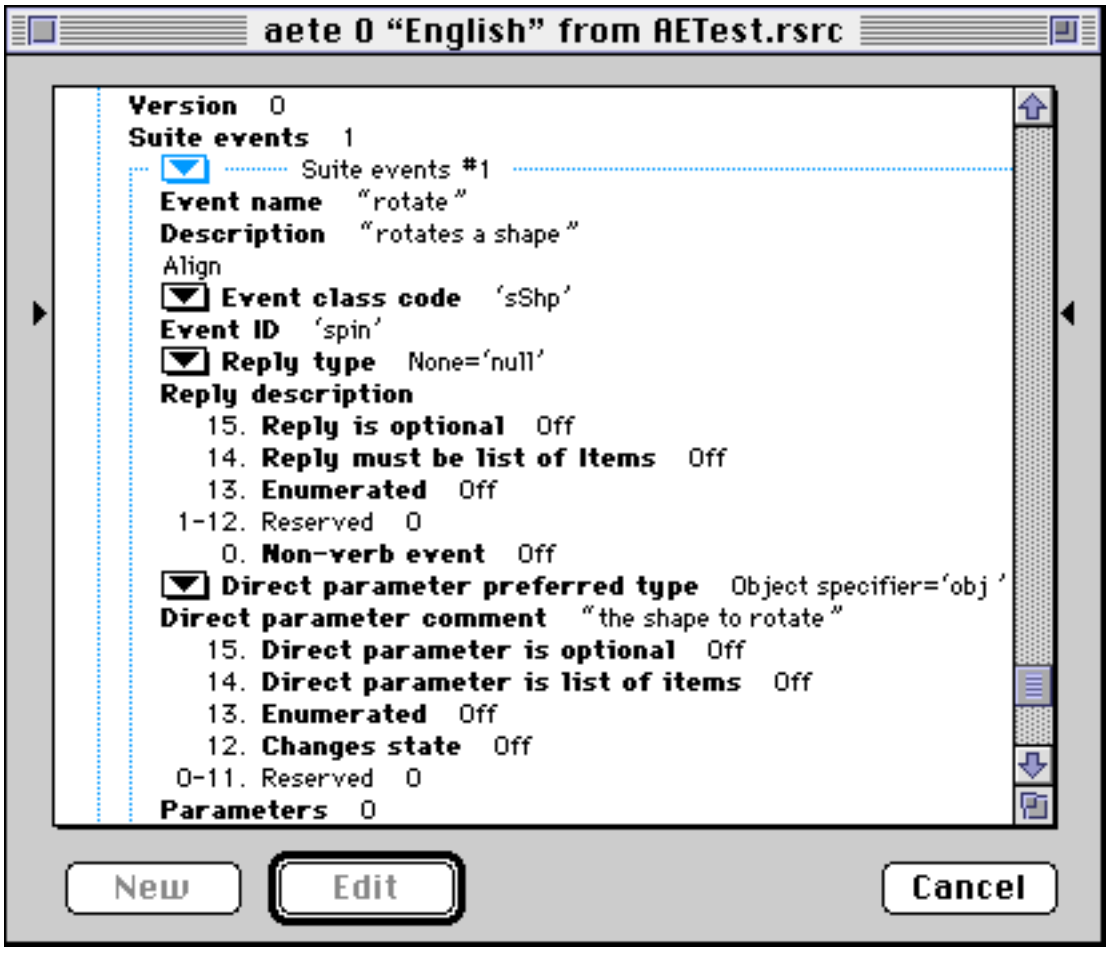
To add a rotate event, move the insertion point to the empty list of events for this suite. Click the **New** button. When you do, all the fields and bits of information related to a single event appear with default values. Set the values to match those shown in Figure 6.6.

Of special importance are the **Event class code** and **Event ID**. You can assign any event class and ID values you like, but remember the values that you use. We recommend you use the values shown in

Figure 6.6 for compatibility with the solution code and further steps in this exercise. However, the codes you use are essentially arbitrary.

For the rotate event, the only parameter is the direct object (e.g. the shape you wish to rotate). The type of direct object parameter should be "object specifier."

Figure 6.6 Adding a rotate event



3. Add a shape class to the new suite.

`'aete' resource` `AETest.rsrc`

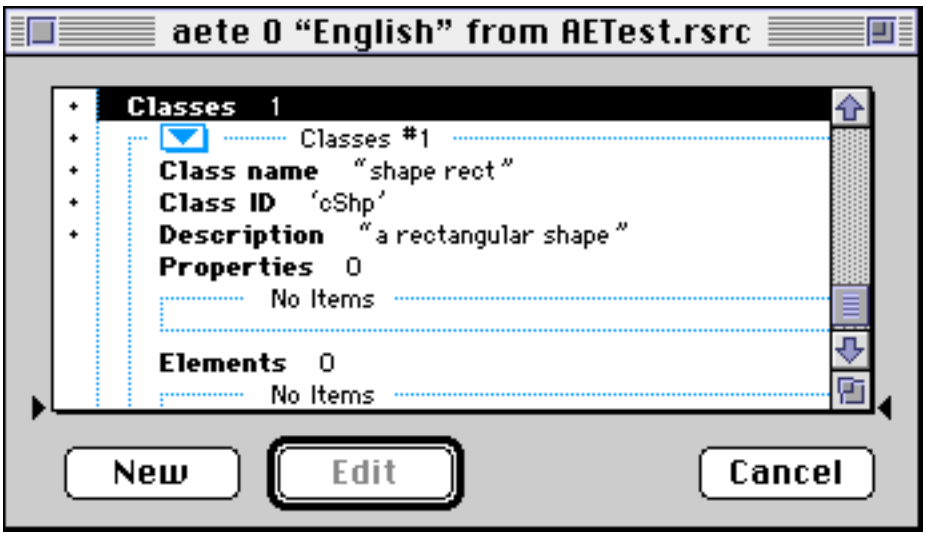
In this step you create the AEOM class for the shape you add to the window. Classes are listed after events in the 'aete' resource. Move the insertion point to the empty list of classes, and click the **New**

button. When you do, all the bits of information associated with a class appear, with default values.

Edit the class name, ID, and description. The ID must match the code you used as an element in the window class, in this case 'cShp.'

When you are through, the shape class should look like Figure 6.7. The properties and elements are empty. That's the next step.

Figure 6.7 The new shape class



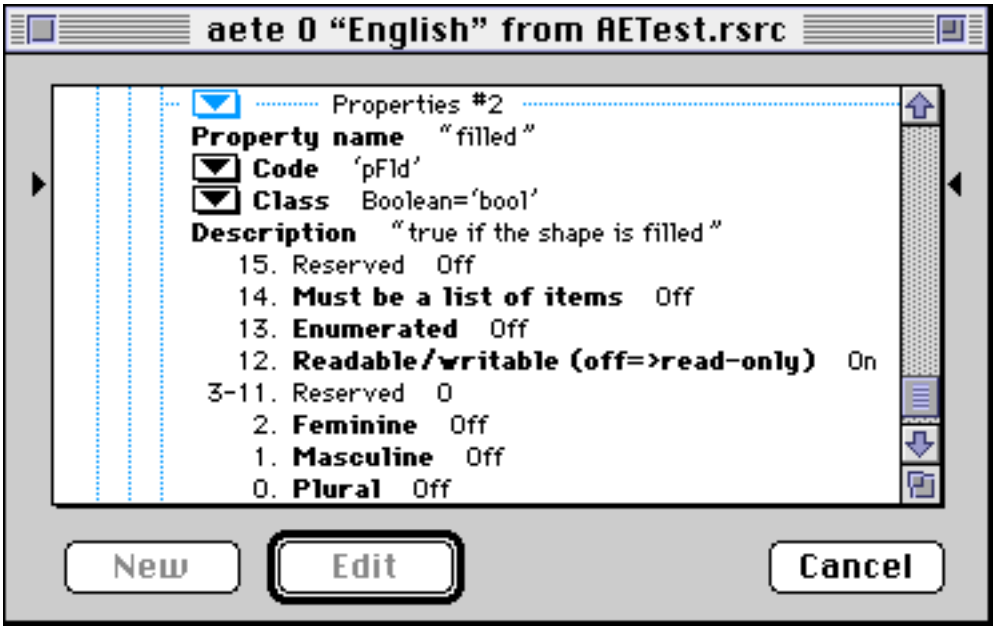
4. Add properties to the new class.

`'aete' resource` `AETest.rsrc`

In this code exercise, you modify the rectangle's line width, and whether or not the shape is filled. The shape class should have two properties: line width and filled. The former should be a short integer, and the latter a boolean.

Move the insertion point to the empty properties list, and click the **New** button. When you do, all the information associated with a property appears in the resource, with default values. Set the values to match those in Figure 6.8.

Figure 6.8 Values for line width and fill properties



Note particularly the **Readable/writable** attribute of the property. Make sure this is set to **On** so that you can write this information.

Repeat the process for the second property, the fill. In this case, all you will specify is whether the rectangle is filled, not what the fill is. All you need for that is a boolean value.

Remember the property ID codes you assigned to each property. You will use these in the code you write.

TIP: One advantage of Rez is that you can use a header file to define constants for things like property ID codes, and then include that file in your source code. Then you don't have to remember.

---

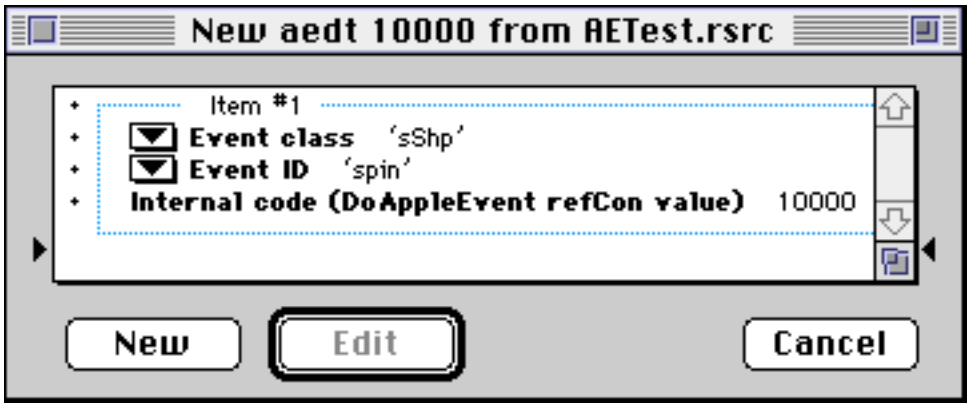
5. Create the 'aedt' resource.

`'aedt' resource` `AETest.rsrc`

In this step you map the event class and ID (from step 2) to a unique long integer that identifies the event inside your code. There is no 'aedt' resource in the file, so you must create one. Use the **New Resource** item on the **Resource** menu to create a new 'aedt' resource. Give it an ID number of 10000.

Then, add a new item to the 'aedt' resource. After you create the item, specify the **Event class** and **Event ID** (from step 2). Then assign an **Internal code** number. In this case, use the number 10000 again. Figure 6.9 shows the result.

Figure 6.9 The new 'aedt' resource



The **Internal Code** is the number you use inside your application to identify the event when you receive it. The file `AETestDef.h` defines the constant `ae_Rotate` to be the value 10000.

You have completed editing the resources. Save your work. You can check your work, if you like, by opening the dictionary in Script Editor. Use the Script Editor's **Open Dictionary** command to open the resource file. The information you set in these steps should appear.

Create a Model Object in the Application

Now you are ready to add the shape class to your implementation of Apple events.

The start code comes with a basic shape class, called CShapeRect, that has members for a rectangle, a line width, and a filled flag. The start code also comes with a CShapeWind class that inherits from LWindow.

You need to make the CShapeRect into an LModelObject.

6. Modify CShapeRect to inherit from LModelObject.

`class declaration` `CShapeRect.h`

In order to support Apple events, an object must inherit from LModelObject. In the class declaration, use public inheritance so that objects of the CShapeRect class descend from LModelObject.

The required code is listed here. As usual, existing code is in italics.

```
class CShapeRect : public LModelObject {
```

The necessary include statement for `LModelObject.h` has been provided for you.

7. Specify a container for the shape when created.

`class declaration` `CShapeRect.h`

`CShapeRect()` `CShapeRect.cp`

When you create an object, you must specify the object's AEOM container. In this case, the container is the window in which the rectangle shape appears.

In the header file, change the prototype for the constructor for CShapeRect to take a single argument. This argument is a pointer to a CShapeWind (CShapeWind *).

```
CShapeRect ( CShapeWind *inSuperModel );
```

In the implementation for the constructor (in the source file), you must receive the new argument (the CShapeWind pointer). Then call the LModelObject constructor. Pass the shape window and the class ID for the shape. The `AETestDef.h` file defines `cShapeRect` for the class ID.

```
CShapeRect::CShapeRect(CShapeWind *inSuperModel)
  : LModelObject(inSuperModel,cShapeRect),
   mFilled(false),mLineWidth(1),mRotateState(0)
```

Steps 6 and 7 make CShapeRect into an Apple-event-savvy object, and prepare for it to be contained by CShapeWind. Next you need to make CShapeWind aware of CShapeRect.

8. Activate the window's submodel list.

`CShapeWind()` `CShapeWind.cp`

In the CShapeWind constructor, activate its use of the built-in submodel list.

```
SetUseSubModelList(true); // to hold the shapes
SetModelKind(cWindow); // this object is window
```

The window will now store a list of its contents (as LModelObjects) in the `mSubModels` data member.

9. Create a shape in response to an Apple event.

`HandleCreateElementEvent()` `CShapeWind.cp`

The object that receives an Apple event telling it to create an element is the container. In this case, that's the CShapeWind object. The class declaration overrides `HandleCreateElementEvent()`. In this step you respond to the event that directs you to create a rectangle.

If this function receives the CShapeRect class ID, you should construct a new CShapeRect. Pass a pointer to the current object (`this`), because it is the containing CShapeWind.

```
switch (inElemClass) {
  case cShapeRect:
    result = new CShapeRect(this);
    break;

  default:
```

This code creates a default object at the end of the current list of elements in the window. In optional steps 15-17, you have the opportunity to improve `HandleCreateElementEvent()` so that it utilizes the "with data," "with properties," and "insert here" parameters of the Apple event.

10. Refresh the screen when contents change.

`AddSubModel()` `CShapeWind.cp`

`RemoveSubModel()` `CShapeWind.cp`

When you add or remove a rectangle from the window, you should refresh the screen. The `AddSubModel()` and `RemoveSubModel()` functions are called by PowerPlant when an element is added or removed from a container. You must override `AddSubModel()` and `RemoveSubModel()`. The existing code calls the inherited version of the function.

PowerPlant adds and removes properties to the `mSubModels` list during execution of an Apple event. You want to refresh only when a CShapeRect is added or removed. You should call `Refresh()`

only if `inSubModel->GetModelKind()` returns the CShapeRect class ID. The required code is shown here.

You want to add the identical code to *both* `AddSubModel()` and `RemoveSubModel()`. In both cases, this code appears after the call to the inherited function.

```
if (cShapeRect == inSubModel->GetModelKind())
{
  Refresh();
}
```

Imaging is not a focus of this code exercise, so drawing routines have been provided for you. `CShapeWind::DrawSelf()` draws each rectangle in the `mSubModels` list. It performs a cast from LModelObject to CShapeRect. This assumes that there is nothing in the window except a CShapeRect.

To make this typecast safer, you could add a test. You could use `dynamic_cast()`, or compare the `model->GetModelKind()` to the class ID you assigned to CShapeRect. The solution code does the latter.

Add Model Properties to the Class

Now that you have made CShapeRect into an LModelObject, you are ready to give it some properties. When you edited the 'aete' you specified "line width" and "filled" properties. Now you need to get and set the C++ members corresponding to those properties.

11. Get properties from an object.

`GetAEProperty()` `CShapeRect.cp`

There are two properties: the line width, and whether the rectangle is filled. In addition, you may receive a message asking for the contents of the object (`pContents`).

The existing code sets up a switch statement, and the default case calls the inherited `GetAEProperty()` function. You create a case for each of the three possible property requests. In each case, call the Toolbox routine `AECreateDesc()` to create the AEDesc containing

the information. Specify the data type, a pointer to the data, the size of the data, and a pointer to the `outPropertyDesc` argument.

The messages received are `pContents`, `pFilled`, and `pLineWidth`. (The latter two are defined in `AETestDef.h`.) For the contents property, provide the bounds of the rectangle, as a `typeQDRectangle`. The data members involved are `mBounds`, `mFilled`, and `mLineWidth`.

```
switch (inProperty) {
  case pContents:
    err = ::AECreateDesc(typeQDRectangle,
                         &mBounds,sizeof(mBounds),
                         &outPropertyDesc);
    FailOSErr_(err);
    break;
  case pFilled:
    err = ::AECreateDesc(typeBoolean,&mFilled,
            sizeof(mFilled),&outPropertyDesc);
    FailOSErr_(err);
    break;
  case pLineWidth:
    err = ::AECreateDesc(typeShortInteger,
              &mLineWidth,sizeof(mLineWidth),
              &outPropertyDesc);
    FailOSErr_(err);
    break;
```

12. Set properties for an object.

`SetAEProperty()` `CShapeRect.cp`

This step is the converse of the previous step. In response to the same message, you retrieve a value from the AEDesc received as an argument by the function, and set the data member.

The existing code sets up a switch statement, and the default case calls the inherited `SetAEProperty()` function. The inherited function simply throws an unknown property exception.

In this step, you create a case for each of the three possible property requests. In each case, you call the appropriate PowerPlant routine in UExtractFromAEDesc to extract information of the correct data

type. The new value is in the parameter `inValue`. Set the correct data member.

The property messages received are `pContents`, `pFilled`, and `pLineWidth`. For the contents property, set the bounds of the rectangle. The data members involved are `mBounds`, `mFilled`, and `mLineWidth`.

Finally, because you are changing the rectangle properties, you should refresh the screen after setting the property.

```
switch (inProperty) {
  case pContents:
    UExtractFromAEDesc::TheRect(inValue,
                                mBounds);
    Refresh();
    break;
  case pFilled:
    UExtractFromAEDesc::TheBoolean(inValue,
                                   mFilled);
    Refresh();
    break;
  case pLineWidth:
    UExtractFromAEDesc::TheInt16(inValue,
                                 mLineWidth);
    Refresh();
    break;
```

The code for `Refresh()` is provided for you. In that code, there is a cast to a CShapeWind. This is safe because the CShapeRect constructor required a reference to a CShapeWind, which was subsequently stored as `mSuperModel` in the CShapeRect.

For a complete implementation of properties, you must also override `GetImportantAEProperties()`. This should call the inherited version first, to build an AERecord with the contents property. Then add the line width and filled properties of your object.

We do not provide the code here to accomplish this task. You can peek at `LModelObject::GetImportantAEProperties()` for hints in the comments for that function. Also look at the solution

code to see how it is done. `HandleClone()` uses this function to determine which properties of your object should be cloned.

Add a Custom Event to the Application

Finally, you need to add an implementation for the rotate event.

13. Identify and handle Apple events.

`HandleAppleEvent()` `CShapeRect.cp`

The object receiving the Apple event is the CShapeRect. As a descendant of LModelObject, it has a `HandleAppleEvent()` function. This function should identify the event and dispatch control to the application code that implements the requested action. In this case, you want to detect the Apple event code for the rotate event.

Remember that you assigned a long integer to this event in the 'aedt' resource. If you followed the solution code, that value is 10000. The file `AETestDef.h` defines `ae_Rotate` as a constant for that value. The long integer identifying the nature of the event is received in the `inAENumber` argument.

Use a switch statement. If you detect a rotate event, call the `Rotate()` function. The `Rotate()` function has been provided for you. For any other event, call the inherited `HandleAppleEvent()`. In this way the superclass's (LModelObject or a descendant) Apple event handling routines get called.

```
switch (inAENumber) {
  case ae_Rotate:
    Rotate();
    break;

  default:
    inherited::HandleAppleEvent(inAppleEvent,
            outAEReply, outResult, inAENumber);
    break;
}
```

In a more complicated event, you might also have to retrieve some required and optional parameters.

14. Build and run the application.

At this point, you can run and debug your code. You have fully implemented basic Apple event support.

When the application builds successfully and runs, an empty window appears. If you look in the **File** menu, there is only one item, **Quit**. You don't want to do that just yet.

There is also a **Script** menu. This menu contains all the sample scripts provided for you. These scripts implement all the application functionality. To perform an operation, choose the corresponding script. You may want to set breakpoints in the code you wrote to see how all the pieces fit together.

WARNING! Don't choose the **delete first rect** script unless there is a rectangle in the window. This sample doesn't handle errors very gracefully, and might crash if you attempt to remove a non-existent object.

Because the window is empty, choose the **make new rect** item. A rectangle appears in the window. Choose the **rotate** item, and the rectangle rotates. Choose the other items, and watch what happens. You can add several rectangles to the window if you wish.

What happens if you have several rectangles and you choose **rotate**? On which rectangle does the script you choose operate? Why?

When you are through exploring, quit the application and read on.

We provided several scripts for you to use. At this point you may want to write your own script that drives the AETest application. For example, use Script Editor or your favorite script authoring environment to write a script that creates three rectangles automatically. Or, set the line width to an arbitrary value. Add your new script or scripts to the Script Menu Items folder. Then launch the application again.

Your script should appear in the **Script** menu. Choose your new script, and see if the application behaves properly.

TIP: By the way, feel free to examine the Script menu code provided for you, and to use that code in your own projects. It demonstrates how easy it is to attach a menu to an application, and how to build that menu out of the contents of a folder.

As you play with the scripts, either your own or those provided for you, you may notice that you cannot write a script that creates a rectangle according to specification. You get a default rectangle every time. A fully-scriptable application should allow the scripter to create an item with complete specifications. That's what we do in the next section.

Improve HandleCreateElementEvent()

You can improve your handling of the create element event to deal with data, properties, and the "insert here" specifier. The solution project provides a very thorough implementation of `HandleCreateElementEvent()`. If you are feeling brave, try to add three additional features to your implementation without peeking! You want to create a rectangle with specified bounds, with specified properties, and in an arbitrary position in the window contents list.

15. Create a rectangle with specified bounds.

`HandleCreateElementEvent()` `CShapeWind.cp`

In this step you modify `HandleCreateElementEvent()` to decode the "with data" parameter and use it to set the new CShapeRect's bounds.

After you successfully create an object, you can then set its properties. You must retrieve the appropriate property from the Apple event. The key for this parameter is `keyAEData`. You can get it using `StAEDescriptor::GetOptionalParamDesc()`. If you succeed (e.g. your descriptor record's `dataHandle` field is not empty), call `SetAEProperty()` to set your new object's content.

(The solution code is in brackets to control the scope of the local StAEDescriptor variables).

```
if (result) {
  {
```

```
    StAEDescriptor data;
    data.GetOptionalParamDesc(inAppleEvent,
                        keyAEData,typeWildCard);
    if (data.mDesc.dataHandle) // has a value
    {
      StAEDescriptor ignore;
      result->SetAEProperty( pContents, data,
                             ignore);
    }
  }
```

16. Create a rectangle with specified properties

`HandleCreateElementEvent()` `CShapeWind.cp`

In this step you modify the same function as in Step 15, but this time you decode the "with properties" parameter.

The key for this parameter is `keyAEPropData`. The parameter is a record that may contain any or all of the contents, fill, and line width properties. You can count the items in the record with the Toolbox routine `::AECountItems()`. You can then iterate through the record with `::AEGetNthDesc()`. Then call your new CShapeRect's `SetAEProperty()` function with each key and descriptor.

```
StAEDescriptor props;
props.GetOptionalParamDesc(inAppleEvent,
                      keyAEPropData,typeAERecord);
if (props.mDesc.dataHandle) { // has a value
  OSErr err;
  long max;
  err = ::AECountItems(props,&max);
  FailOSErr_(err);
  for( long i = 1; i <= max; ++i) {
    DescType         theKeyword;
    StAEDescriptor   theValue, ignore;
    err = ::AEGetNthDesc( props, i,
          typeWildCard, &theKeyword, theValue);
    FailOSErr_(err);
    result->SetAEProperty( theKeyword,
                           theValue, ignore);
  }
}
```

17. Move the shape to the correct position in mSubModels.

`HandleCreateElementEvent()` `CShapeWind.cp`

In this step you specify the position of the new rectangle in the list of existing rectangles. The arguments `inInsertPosition` and `inTargetObject` tell you the destination location. The value of `inInsertPosition` can be `kAEBeginning`, `kAEEnd`, `kAEBefore`, `kAEAfter`, or `kAEReplace`. In the latter three cases, `inTargetObject` is the CShapeRect that your new object goes before, goes after, or replaces.

You can use these arguments to calculate the target position of the new object in the LList of contents in the window. Remember, the `mSubModels` data member is an LList object. You'll need the current position of the new object. You can use `LList::FetchIndexOf()`. By default, the initial target position should be the current position.

You must then calculate the correct target position. Switch on the value of `inInsertPosition`. As you know, LList is a one-based array, so position 1 is the beginning. The constant `arrayIndex_Last` specifies the last position. In the before, after, or replace cases, use `FetchIndexOf()` again to get the position of the `inTargetObject`. Set the target position appropriately (add 1 to the `inTargetObject`'s position to go after `inTargetObject`).

Finally, move the object to the correct position in the list. You do this using `LList::MoveItem()`. If you are replacing an existing rectangle, don't forget to delete it.

```
Int32 currentPosition =
                mSubModels->FetchIndexOf(&result),
targetPosition = currentPosition;
switch (inInsertPosition) {
  case kAEBeginning:
    targetPosition = 1;
    break;

  case kAEEnd:
    targetPosition = arrayIndex_Last;
    break;
```

```
    case kAEBefore:
    case kAEReplace:
      targetPosition =
        mSubModels->FetchIndexOf(&inTargetObject);
      break;

    case kAEAfter:
      targetPosition = 1 +
        mSubModels->FetchIndexOf(&inTargetObject);
      break;
  }
  mSubModels->MoveItem( currentPosition,
                        targetPosition);
  if (inInsertPosition == kAEReplace)
  {
    delete inTargetObject;
  }
```

Congratulations, you're done!

If you've done everything correctly, you should be able to write scripts that specify the properties and position of a rectangle when it is created. Go ahead, give it a try. If your scripts do not work correctly, fire up the debugger and trace the execution of your code.

Tracing execution with the debugger is a good idea, even if your code does work. By tracing the Apple event handling process, you'll learn more about how LModelDirector, LModelObject, and LModelProperty work than could be discussed here.

As we mentioned at the beginning of this chapter, Apple events in PowerPlant is a very big subject. This chapter is intended to get you over the initial hump. You now know how to create a basic scriptable application in PowerPlant.

There is plenty of room for further learning and exploration. As you apply these concepts to your own projects, you will learn much more about PowerPlant's handling of Apple events. Whatever direction you choose, good luck, and have fun. PowerPlant should help make your task easier.

7

Networking in PowerPlant

This chapter discusses how to use the PowerPlant network classes to create a network-savvy application.

Introduction to Networking in PowerPlant

In today's working environment more and more software applications are becoming collaborative in nature. Individuals and groups working together require applications that can communicate with other applications—sometimes across great distances over the global Internet.

The Internet is the largest computer network in the world and connects millions of computers in hundreds of countries. These computers, although running different operating systems and applications, can all "speak" to one another by means of a standard protocol known as the Internet Protocol (IP). IP provides the basis for connectionless, best-effort data packet delivery between computers.

The Transmission Control Protocol (TCP) builds upon the functionality of IP by providing reliable, full-duplex, connection-oriented, stream communications. That is, once the two ends of a TCP connection are established, data can be sent and received between the two, simultaneously, until the connection is closed or broken. Together these protocols are often referred to as TCP/IP and they form the basis for Internet communication.

The Mac OS supports two mechanisms for implementing TCP/IP:

- MacTCP
- Open Transport

The PowerPlant network classes support both MacTCP and Open Transport in order to provide the most compatible and optimized implementation depending on the currently running system software. The classes provide numerous high-level functions so you do not have to worry about the details of MacTCP or Open Transport directly. When you implement your application using the PowerPlant network classes, you need not be concerned if the user has MacTCP or Open Transport installed on their computer—the classes will handle the details for you.

The PowerPlant network classes also allow you to perform simple UDP implementations. UDP is the User Datagram Protocol and allows you to send data to a remote computer without having to maintain a connection to that computer. This is discussed briefly in the section entitled "Connectionless Datagram Communications" on page 243.

The topics in this chapter include:

- Networking Strategy—PowerPlant's approach to networking
- Networking Classes—a detailed look at the PowerPlant classes involved in network support
- Implementing a Network-Savvy Application—how to implement simple networking in your application
- Summary of Networking in PowerPlant
- Code Exercise for Networking

NOTE: The PowerPlant network classes that are included with CodeWarrior 11 or later are not compatible with versions included before CodeWarrior 11. Any code that you've written prior to CodeWarrior 11 that uses the PowerPlant network classes will need to be updated to use the new architecture. See the PowerPlant network classes release notes on your CodeWarrior CD for more information on specific changes between versions of the classes.

Where to Learn More About Networking

This chapter does not teach you the intricacies of data communications, using TCP/IP with MacTCP or Open Transport, or any of the standard Internet protocols such as HTTP (World Wide Web HyperText Transfer Protocol), FTP (File Transfer Protocol), or SMTP (Simple Mail Transfer Protocol). It also assumes that you are familiar with basic communications techniques in general. You should note that writing communications software is not for the faint of heart, even with great tools like the PowerPlant network classes. There are many intricate details to writing robust communications code that only experience can teach you. For more information regarding these topics, you may wish consult the following books and documentation. This material will help you learn how to implement Internet-savvy applications.

Comer, Douglas E. *Internetworking With TCP/IP, Volume I, Principles, Protocols, and Architecture*. Prentice Hall. ISBN 0-13-216987-8

MacTCP documentation available from Apple Computer, Inc. at ftp://ftp.apple.com/devworld/Development_Kits/MacTCP/

Open Transport documentation available from Apple Computer Inc., and also on your CodeWarrior CD.

InterNIC at http://rs.internic.net/

Internet Engineering Task Force at http://www.ietf.cnri.reston.va.us/home.html

The WebStar site also has useful information for developers who are interested in writing Internet code at http://www.starnine.com/

The Netscape site at http://home.netscape.com/

Software Requirements

The PowerPlant network classes make use of either the MacTCP or Open Transport system software. Given this, before you can use the classes in your program you must have this system software installed and configured properly. You must also have some type of

TCP/IP connection such as directly to an Ethernet network or to an Internet access provider using PPP (Point-to-Point Protocol) software.

How to configure your machine and network is outside the scope of this document. However, there are plenty of places to learn more. If you do not know where to turn, consider purchasing the Apple Internet Connection Kit from Apple Computer, Inc. This kit includes everything you need to get connected to the Internet.

The PowerPlant network classes also require the PowerPlant Threads classes which in turn require the Thread Manager from Apple Computer, Inc. If you are running the most recent version of the Mac OS System Software then you most likely have the Thread Manager already installed. If not, you can obtain a copy of the Thread Manager from the Apple Computer, Inc. web site at http://www.apple.com/

It should also be noted that the PowerPlant network classes require the use of the Open Transport Client Developer libraries version 1.1.1 or later. These libraries can be found on your CodeWarrior CD or on Apple's Open Transport web site at http://devworld.apple.com/dev/opentransport/. They are installed automatically when you use the installer on your CodeWarrior CD.

NOTE: If you are running Open Transport 1.1 or later, the Open Transport API will be called from 68k code when running on a PowerPC. If you are running versions of Open Transport prior to 1.1, MacTCP will be called from 68k code when running on a PowerPC. This is due to the fact that versions of Open Transport prior to 1.1 were never shipped for 68K Macintosh computers, therefore it is assumed that MacTCP is available instead. For the adventurous, you can set or reset the OPENTPT_ON_68K compile option, but the default is to use Open Transport whenever possible. You should also understand that older code written using MacTCP only should run with little or no modification under Open Transport.

Networking Strategy

The PowerPlant network classes implement both MacTCP and Open Transport compatibility while providing a single, common API to TCP/IP communication. This API shields you from the intricacies of both MacTCP and Open Transport so you can concentrate on the functionality of your application and not low-level communications details. The API resembles the form and function of Open Transport, the newest communications technology for the Mac OS.

NOTE: The PowerPlant network classes were designed to be used either within the PowerPlant framework or without it. Therefore, you can include the functionality provided by the network classes in your application without making use of any of the other classes in PowerPlant, except the required PowerPlant Threads classes.

Generic Network Interface

When implementing network support in your application, you need only concern yourself with a small number of classes to provide support for establishing connections, sending data and receiving data. These are the basic functions that are necessary for Internet communications. Collectively these classes are known as the generic network interface and consist of:

- UNetworkFactory
- LInternetAddress
- LTCPEndpoint
- LUDPEndpoint

UNetworkFactory is a utility class that creates network endpoints and mappers using the best configuration (MacTCP or Open Transport) given the current running system software.

LInternetAddress represents both IP and DNS style Internet addresses. It will automatically map between DNS style and IP style addresses for you as necessary.

NOTE: IP addresses refer to numbered addresses, such as 127.0.0.1. DNS addresses refer to named addresses such as www.metrowerks.com.

LTCPEndpoint represents a TCP/IP style network connection. It establishes a connection between your application and another on a remote computer using the Transmission Control Protocol. When created it will use the best configuration (MacTCP or Open Transport) given the current running system software.

LUDPEndpoint represents a UDP style network connection. It establishes a connection between your application and another on a remote computer using the User Datagram Protocol. When created it will use the best configuration (MacTCP or Open Transport) given the current running system software.

Other Classes

Other classes exist in the PowerPlant network classes. However, they implement low-level, internal methods that are not discussed in this chapter. You need not be concerned with them in order to implement networking in your PowerPlant application. You should feel free to explore them as your understanding of networking increases, however, the Generic Network Interface classes shield you from the details within them.

Strategic Summary

As mentioned earlier, the PowerPlant network classes make use of the PowerPlant Threads classes. In this way, your entire network implementation is "threaded" and therefore is extremely optimized for communications. By using threads, the network classes are able to take advantage of all the features that the Thread Manager and PowerPlant Threads classes have to offer. Other advantages include:

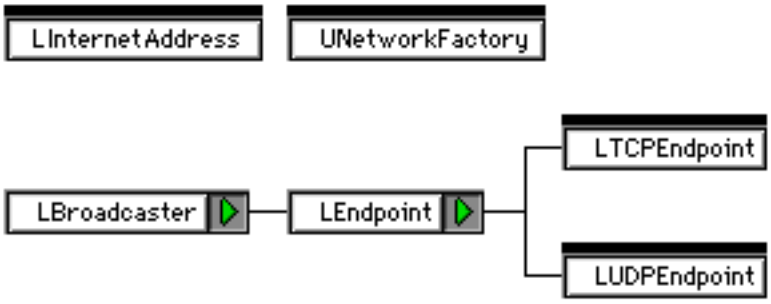
- A highly optimized architecture for asynchronous network events
- The ability to have multiple communication sessions active with little or no degradation in throughput

- The ability to abort any network operation
- The automatic timing out of any network operation
- Simple implementation of protocols self-contained in the thread's `Run()` method

Networking Classes

Now that you have an idea of what the individual network classes do, let's take a more in-depth look at the functionality of each. Figure 7.1 illustrates the class hierarchy that you will be concerned with when implementing networking in your PowerPlant application.

Figure 7.1 The classes that you will use directly



Classes discussed in this section include:

- UNetworkFactory
- LInternetAddress
- LTCPEndpoint
- LUDPEndpoint

UNetworkFactory

The UNetworkFactory utility functions allow you to easily create TCP endpoints and mappers based on the current running system software. Calling simple functions in this utility class automatically choose MacTCP or Open Transport support transparently to you and your program.

UNetworkFactory includes only a few but very important functions as follows:

Table 7.1 Some vital UNetworkFactory functions

Function	Purpose
`CreateTCP-Endpoint()`	creates the best TCP/IP endpoint object based on the current running system software (MacTCP or Open Transport)
`CreateUDP-Endpoint()`	creates the best UDP endpoint object based on the current running system software (MacTCP or Open Transport)
`CreateInter-netMapper()`	creates the best name mapper object based on the current running system software (MacTCP or Open Transport)
`HasTCP()`	returns true if a TCP/IP protocol stack is installed
`HasOpenTrans-port()`	returns true if Open Transport is installed
`HasMacTCP()`	returns true if MacTCP is installed

NOTE: It is unlikely that you will ever need to create a mapper object (using `CreateInternetMapper`()) yourself since the endpoint objects (discussed below) can accept address objects as arguments when connecting to a remote computer. Also, the endpoint objects will perform any necessary DNS lookups automatically for you. `CreateInternetMapper`() is provided as a convenience if you need to perform DNS name or address lookups without maintaining a connection to the remote computer.

LInternetAddress

LInternetAddress represents both IP and DNS style Internet addresses. It will automatically map between DNS style and IP style addresses for you as necessary.

LInternetAddress includes numerous constructors allowing you to pass various parameters to the object including a numbered address, a named address and a port number. It also contains many useful functions as follows:

Table 7.2 Some vital LInternetAddress functions

Function	Purpose
`GetIPDescriptor()`	converts the IP address into a dotted decimal format and returns it as a string—optionally appends the port number (i.e.: 127.0.0.1:80)
`GetDNSDescriptor()`	returns the DNS name of the host computer—optionally appends the port number (i.e.: www.metrowerks.com:80)—ensures a DNS lookup has been performed
`GetIPAddress()`	returns the 32-bit IP address of the host—optionally returns the dotted decimal format as a string
`GetDNSAddress()`	returns a string representing the address of the host computer—may be in dotted decimal format—does not ensure a DNS lookup has been performed
`SetIPAddress()`	allows you to set the 32-bit host address
`SetDNSAddress()`	allows you to set the DNS address as a string—either by name or as dotted decimal

Function	Purpose
`GetHostPort()`	returns the port of the host address
`SetHostPort()`	allows you to set the port of the host address
`MakeOTIPAddress()`	returns the host address as an Open Transport TNetbuf/InetAddress structure—for use in advanced Open Transport calls that you may make outside of the PowerPlant network classes
`MakeOTDNSAddress()`	returns the host address as an Open Transport TNetbuf/DNSAddress structure—for use in advanced Open Transport calls that you may make outside of the PowerPlant network classes
`Clone()`	returns a copy (via operator new) of the LInternetAddress object

NOTE: Something that we have yet to touch on is the fact that all TCP connections occur on ports. There are over 65000 ports to choose from on any one computer. Some ports are “well known” or “reserved” such as port 80 which is used by HTTP (World Wide Web) servers. Servers usually “listen” for incoming connections on a particular port. Outgoing connections, however, usually use any port that is available at the time the endpoint binds. Binding is the task of telling the computer that you want to make use of a particular port. If the IP or DNS address is the “street name” then the port number being used is your “house number” on that street. As long as someone knows how to get to your street, and knows what number your house is, they can contact you.

LTCPEndpoint

LTCPEndpoint forms the basis for TCP/IP networking in PowerPlant. This is a simple class that inherits from the abstract base class LEndpoint. When you call the UNetworkFactory function `CreateTCPEndpoint()` an LTCPEndpoint will be created for you automatically in the form of either an LOpenTptTCPEndpoint or an LMacTCPTCPEndpoint depending on the current system software. Either way, the calls that you make to bind, connect, transfer data and disconnect will be the same.

TCP/IP is used for many session-oriented protocols such as HTTP (World Wide Web HyperText Transfer Protocol), FTP (File Transfer Protocol), POP (Post Office Protocol) and SMTP (Simple Mail Transfer Protocol).

TIP: Be sure to explore the PowerPlant Internet classes, found on your CodeWarrior CD. These classes implement many popular protocols on top of the PowerPlant network classes including HTTP, FTP, POP and SMTP. Not only do they offer your PowerPlant application these popular services, with a minimum of effort on your part, but they are also an excellent example of implementing session-oriented protocols using the PowerPlant network classes.

This section covers those functions that you are most likely to encounter directly. Use the *PowerPlant Reference* for detailed and complete technical information.

LTCPEndpoint and its related classes have numerous member functions that you will find useful in order to allow you to bind to a local port, connect to a remote computer and send and receive data.

Table 7.3 Some vital LTCPEndpoint functions

Function	Purpose
`LTCPEndpoint()`	constructor
`Local Address Configuration`	
`Bind()`	reserve a local TCP port for the connection
`Unbind()`	release a previously bound local TCP port
`Connection Establishment (Clients)`	
`Connect()`	connect to a remote computer
`Disconnect()`	terminate the connection to a remote computer via an orderly disconnect—blocks the thread until the remote computer has shut down the connection and therefore does not allow further data transfer via the connection
`SendDisconnect()`	terminate the connection to a remote computer via an orderly disconnect—returns immediately and allows further reception of data via the connection
`AbortiveDisconnect()`	terminate the connection to a remote computer without waiting for the remote computer to acknowledge
`AcceptRemoteDisconnect()`	accept an incoming disconnect request from a remote computer
`Passive Connections (Servers)`	
`Listen()`	obtain information about an incoming connection from a remote computer
`AcceptIncoming()`	accept an incoming connection from a remote computer

Function	Purpose
`RejectIncoming()`	reject an incoming connection from a remote computer
`Host Addresses`	
`GetLocalAddress()`	returns the address of the local computer as an LInternetAddress object
`GetRemoteHost-Address()`	returns the address of the remote computer as an LInternetAddress object
`Sending Data`	
`Send()`	send data to a remote computer—accepts a void pointer to data and size argument
`SendData()`	see Send()—with optional expedited flag and timeout
`SendPStr()`	send data to a remote computer—accepts a pointer to a pascal string
`SendCStr()`	send data to a remote computer—accepts a pointer to a C string
`SendHandle()`	send data to a remote computer—accepts a Macintosh Memory Manager Handle
`SendPtr()`	send data to a remote computer—accepts a Macintosh Memory Manager Pointer
`Receiving Data`	
`Receive()`	receive data from a remote computer—accepts a void pointer to a buffer and size argument
`ReceiveData()`	see Receive()—with optional expedited flag and timeout

Function	Purpose
`ReceiveDataUntil-Match()`	see ReceiveData()—with optional match character
`ReceiveLine()`	see Receive()—with optional "use Line-Feed" flag and timeout
`ReceiveChar()`	receive a single character from a remote computer—with optional timeout
`Connection Status`	
`GetState()`	return the current state of the endpoint
`Receive Configuration`	
`GetAmountUnread()`	return the number of bytes of unread data on the endpoint
`Acknowledgment of Sent Data`	
`AckSends()`	enable the acknowledgment of sent data mechanism for the endpoint
`DontAckSends()`	disable the acknowledgment of sent data mechanism for the endpoint
`IsAckingSends()`	return if the acknowledgment of sent data mechanism for the endpoint is enabled or disabled
`QueueSends()`	enable the queuing of data to be sent mechanism— which returns control to the caller immediately after calling a send function
`DontQueueSends()`	disable the queuing of data to be sent mechanism— which returns control to the caller once the data has actually been sent
`IsQueuingSends()`	return if the queuing of data to be sent mechanism is enabled or disabled

Function	Purpose
`Miscellaneous`	
`AbortThreadOpera-tion()`	aborts the current threaded operation—accepts the thread to abort as the argument

One item to keep in mind when using the `SendData` function is that the 65K single buffer size limitation of MacTCP has also been implemented in the Open Transport code of the classes. This was done to keep the API the same for both MacTCP and Open Transport without causing subtle differences between the implementations. If you must send more than 65K of data in one burst, you should split it up into multiple buffers before sending.

NOTE: For consistency and simplicity of end user code design, threads are blocked after a ReceiveData() call until there is data on the endpoint. Thus, you can create simple receive loops that block the thread without having to do your own “Receive - If (no data) Yield()” type loop as was required in previous versions of the PowerPlant network classes. The downside of this functionality is that since the receive has been blocked you are most likely to get hit with unexpected messages (usually disconnects) while in this state. Since you have asked specifically for a receive, anything that is not a typical completion for receive gets thrown back to you as an exception. Most of the time, you can expect this to be a disconnect or an orderly disconnect. While these are not necessarily “error conditions” in terms of the connection, they are unexpected events and get thrown back to you as such.

LUDPEndpoint

LUDPEndpoint forms the basis for sessionless (connectionless) networking in PowerPlant. This is a simple class that inherits from the abstract base class LEndpoint. When you call the UNetworkFactory function `CreateUDPEndpoint()` an LUDPEndpoint will be creat-

ed for you automatically in the form of either an LOpenTptUDPEndpoint or an LMacTCPUDPEndpoint depending on the current running system software. Either way, the calls that you make to bind and transfer data will be the same.

UDP is used for many sessionless protocols such as NTP (Network Time Protocol) and for implementing echo, ping and traceroute functionality.

This section covers those functions that you are most likely to encounter directly. Use the *PowerPlant Reference* for detailed and complete technical information.

LUDPEndpoint and its related classes have numerous member functions that you will find useful in order to allow you to bind to a port and send and receive data.

Table 7.4 Some vital LUDPEndpoint functions

Function	Purpose
`LUDPEndpoint()`	constructor
`UDP datagram messaging`	
`Bind()`	reserve a local UDP port for the "connection"
`Unbind()`	release a previously bound local UDP port
`Host Addresses`	
`GetLocalAddress()`	returns the address of the local computer as an LInternetAddress object
`GetRemoteHost-Address()`	returns the address of the remote computer as an LInternetAddress object
`Sending Data`	
`SendPacketData()`	send data to a remote computer
`Receiving Data`	

Function	Purpose
`ReceiveFrom()`	receive data from a remote computer
`Connection Status`	
`GetState()`	return the current state of the endpoint
`Acknowledgment of Sent Data`	
`AckSends()`	enable the acknowledgment of sent data mechanism for the endpoint
`DontAckSends()`	disable the acknowledgment of sent data mechanism for the endpoint
`IsAckingSends()`	return if the acknowledgment of sent data mechanism for the endpoint is enabled or disabled
`QueueSends()`	enable the queuing of data to be sent mechanism— which returns control to the caller immediately after calling the `SendPacketData`() function
`DontQueueSends()`	disable the queuing of data to be sent mechanism— which returns control to the caller once the data has actually been sent
`IsQueuingSends()`	return if the queuing of data to be sent mechanism is enabled or disabled
`Miscellaneous`	
`AbortThreadOpera-tion()`	aborts the current threaded operation— accepts the thread to abort as the argument

TIP: For more information on how to implement communications protocols, handle communications errors robustly, and take advantage of communications programming tactics, consult a communi-

cations text-book. See also “Where to Learn More About Networking” on page 219.

Implementing a Network-Savvy Application

Implementing network support in your application is relatively simple when using the PowerPlant network classes because the classes shield you from the details of the OS networking implementation. Although there are numerous ways to use these classes, one example of implementing simple network support follows.

When you implement network applications you write a client, a server, or a client/server application. A client application is one that requests information from a server. A server application is one that supplies information to a client or clients. A client/server, or peer-to-peer application, includes both client and server functionality in the same application. A chat program that allows two individuals to communicate directly with each other is an example of a client/server application.

In this section we discuss the steps required to build both the client and server side of a network application. Because the PowerPlant network classes make use of the PowerPlant Threads classes you may wish to review the chapter which discusses them before continuing.

One thing to keep in mind is that most of the functions dealing with binding, connecting, sending and receiving data are all being called from within a thread. Many times the functions that you call will block the thread until the asynchronous task completes. This makes it extremely easy to implement protocols and client architectures from within the `Run()` method of your thread.

Each topic in this section reflects a task you may need to perform when writing your own network applications. This section includes the following topics:

- Creating a Client
- Obtaining an Address

- Creating a Client Endpoint
- Binding to a Local Port
- Connecting to a Server
- Sending Data
- Receiving Data
- Disconnecting from a Server
- Handling a Disconnect Request
- Creating a Server
- Listening for Incoming Connections
- Responding to Incoming Connections

More specialized topics follow including:

- Implementing Threads
- Connectionless Datagram Communications

Creating a Client

A client is the end of the connection that initiates the communication between itself and a server. If you've ever used any Internet software such as a browser, file transfer application, news reader, etc. then you have used a client application.

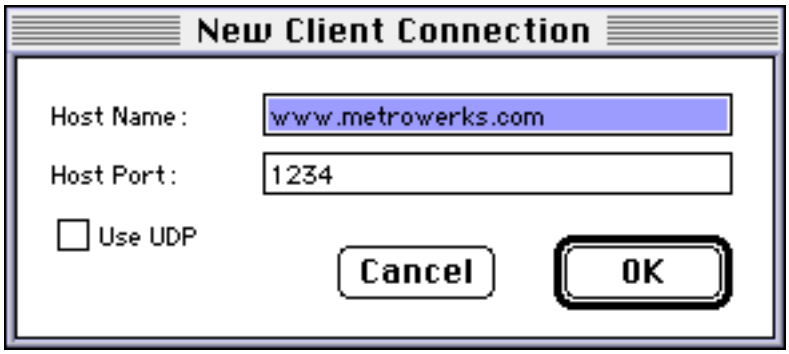
Before you implement your client application you need to know which protocols you will be supporting. In order to write a program that is useful it must support a protocol above TCP/IP such as HTTP, FTP, or a protocol of your own design.

TIP: You may choose to use the PowerPlant Internet classes to provide your protocol support. The PowerPlant Internet classes support many common protocols used on the Internet today. Review the PowerPlant Internet classes documentation for complete information.

You first need to create a client class that creates and maintains any user interface elements you may need—such as a window and con-

trols. Once your window is created you can offer the user some simple controls to allow them to pick the address of a remote computer to connect to.

Figure 7.2 A sample client window



Obtaining an Address

In order to connect to a remote computer you must know its address on the network. Addresses come in a variety of forms, as discussed earlier. Normally you request the remote address from the user of your application as a string of text, either dotted decimal or DNS name, and simply create an LInternetAddress object from it. This is used later to connect to the remote computer.

Creating a Client Endpoint

Before you can communicate with the remote computer you must create an endpoint. Every connection between a client and server has two endpoints, one on each end of the connection. You create your client endpoint by calling `UNetworkFactory::CreateTCPEndpoint()`. If you were creating a connectionless UDP endpoint you would call `UNetworkFactory::CreateUDPEndpoint()` instead.

Binding to a Local Port

Now that you have created your client endpoint you need to bind it to a local port in order to communicate. As you may recall, your cli-

ent machine has an Internet address but also has thousands of possible ports on the machine that it can use to communicate through. By binding to one of these local ports you are making it possible for the server to find your application easily amongst any other applications that might be simultaneously using IP on your computer.

To bind to a local port you need to create an LInternetAddress object. Pass zero for both the host address and the host port in the constructor. These parameters mean that you will use any available port as the outgoing communications "channel." Because the client is initiating the connection, the port number really doesn't matter.

Pass the LInternetAddress object to `LEndpoint::Bind()` to perform the bind. You should also pass zero for the `inListenQueueSize` function parameter to `Bind()`. You pass a different value as `inListenQueueSize` when you create a server. Servers are interested in listening for incoming connections, clients are not. The thread that is handling your client side of the equation will be blocked until the bind completes.

Connecting to a Server

To actually connect to the remote endpoint, you simply need to pass the LInternetAddress object to the endpoint via its `Connect()` member function.

At this point the thread will be blocked once again and upon it being resumed the connection will have either taken place or failed. Assuming the connection was made, you can continue to send and/or receive data from within your thread. If the connection failed your try/catch block will have caught any exceptions and you can handle them accordingly.

NOTE: Servers refuse connections for a variety of reasons. For example, if an FTP server has too many users transferring files at the moment you connect, it refuses your connection. It may also refuse your connection because you entered the wrong password or don't have access to the server. If you try to connect to a machine that you think is running a particular server but it is not, your

connection will also be refused by the TCP software on that machine because there are no listeners on that particular port.

Sending Data

To send data to the remote computer you merely need to call `any one of the Send()` member functions depending on the type of data you are sending. You can rest assured that the data will arrive at its destination in the majority of cases because TCP/IP does its best to guarantee this. If you are using a UDP endpoint, however, the data is not guaranteed to arrive and you will need to call the UDP specific send function instead.

Receiving Data

To receive data from the remote endpoint you merely need to call `any one of the Receive()` member functions depending on the type and size of data you are receiving. If you are using a UDP endpoint, however, the you will need to call the UDP specific receive function instead.

Disconnecting from a Server

If you wish to initiate a disconnect, simply call either the `Disconnect()` or `SendDisconnect()` member functions. In most cases you will just need to call `Disconnect()` when you are ready to close a connection. After calling the `Disconnect()` function the thread will block and upon being resumed you should call `Unbind()` to release the local port you bound to earlier.

NOTE: If you are connecting via UDP instead of TCP, remember that you need not (and can not) call `Connect()` and `Disconnect()`. Otherwise, using UDP is similar to TCP in the sense that you bind, send, receive and unbind.

Handling a Disconnect Request

If, at any time in your threaded send and receive loop, you receive a `OrderlyDisconnect_Error` message you should call `AcceptRemoteDisconnect()` immediately and continue by deleting the local endpoint. This message is telling you that the remote endpoint initiated a disconnect and you should oblige.

WARNING! Sending data after receiving a `OrderlyDisconnect_Error` message is risky business. Because the remote endpoint may have already closed its end of the connection or may ignore incoming data, the data may never be received.

Creating a Server

A server is the end of the connection that listens for incoming requests from one or more clients. Servers usually offer information that the client can request once connected. Servers are more difficult to write than clients in most cases.

Before you implement your server application you need to know which protocols you will support. A useful application must support a protocol above TCP/IP such as HTTP, FTP, or a protocol of your own design.

NOTE: When writing a server, you may opt to have a server class and a responder class. The server class listens for incoming connections. When one occurs, the server creates a responder object that completes the connection and handles the exchange of data between the client and itself. The responder can function very similarly to the client class mentioned above. An exception is that it need not call `Bind()`, because when it connects by calling `AcceptIncoming()` the bind happens transparently.

You first need to create a server class that listens for incoming connections, and a responder class that responds to these connections.

Your responder class is almost identical to your client class in many respects, because it handles the exact same protocol. The only major difference is in the way it connects, instead of initiating a connection by calling `Connect()`, it accepts an incoming connection request, discussed below.

Listening for Incoming Connections

Servers spend most of their time listening for incoming connection requests. When a server receives a request it can decide whether or not to accept the request based on the IP or DNS address of the computer that the request has originated from, the number of users currently logged into the server, or other criteria that you choose.

To listen for incoming connection requests your server object should first create the server endpoint by calling `UNetworkFactory::CreateTCPEndpoint()`.

Secondly, create an LInternetAddress object. Pass in the port number that you want to listen to for incoming connection requests. For example, if you are writing an HTTP (World Wide Web) server you will use "well-known-port" number 80.

Next, pass the LInternetAddress object to the `Bind()` member function. Your server thread will be blocked and upon it being resumed you will be bound to the port you specified. You should pass any number greater than zero for the `inListenQueueSize` function parameter when calling `Bind()`. This tells TCP/IP that you want to listen for up to `inListenQueueSize` incoming connections at once. Therefore, while one connection is being serviced, others will not be turned away.

NOTE: You should not arbitrarily set the `inListenQueueSize` function parameter. For each connection you offer to service, more RAM is required for use by your application and the TCP/IP subsystem. You should experiment with the number of connections

that best suits your needs and memory requirements. You may also wish to allow the user to set this value via a preference setting but remember that they may also need to increase the memory partition of your application as well.

Assuming the bind completed with no errors, your server object is now listening for incoming connections.

Responding to Incoming Connections

Incoming connection requests are flagged by the `T_LISTEN` message. When your server object receives a `T_LISTEN` message you can evaluate the incoming connection request to see if you would like to service it. Immediately after you receive the `T_LISTEN` message you should call `Listen()` to inform the server endpoint that you will be dealing with the latest incoming connection request.

The easiest way to manage this in your thread is to simple `Suspend()` your thread once the bind has completed successfully. Upon receiving a `T_LISTEN` message your thread will be automatically resumed. You can then call the `Listen()` member function of the server endpoint.

If you choose to reject the incoming connection request, after calling `Listen()`, you simply need to call the `RejectIncoming()` member function of the server endpoint and return to waiting for other `T_LISTEN` messages.

Assuming you choose to accept the incoming connection request, you should create a responder object and pass the endpoint on to it. This forces the responder to connect to the remote requester.

Your responder first creates another endpoint using `UNetworkFactory::CreateTCPEndpoint()`. Once created, the responder's thread calls the server endpoint's `AcceptIncoming()` member function, passing in the newly created responder endpoint as the `inEndpoint` function parameter. This is an extremely important step and can be considered the "hand-off." The server endpoint is handing off the connection to the responder endpoint.

The responder thread will then be blocked and will resume when the hand-off is completed. From this point on the responder can act as if it is a client, following the protocol as defined. The responder is now connected to the client on the remote computer and can freely send and receive data with it.

TIP: When writing a peer-to-peer application (such as a chat program) where the two ends function as both a client and a server, you may opt to derive your responder and client classes from a common base class. In many cases each object, once connected, must handle the exact same protocol as the other with very few, if any, differences.

Implementing Threads

Threaded implementations can be extremely useful when dealing with query/response protocols such as POP or SMTP. That is, a protocol that simply sends a query and then awaits a response is a good candidate for a thread-based implementation.

As mentioned, the PowerPlant network classes depend on the PowerPlant Threads classes in order to function. The heart of your network applications will contain threads that do the majority of the network tasks such as binding, connecting, sending, receiving, disconnecting and unbinding. The internal implementation of the PowerPlant network classes depend on this and make use of blocking,. suspending and resuming of your threads in order to implement an elegant and optimized networking architecture.

For complete examples of how to implement your threaded network implementation, see the SimpleClient and SimpleServer examples on your CodeWarrior CD and their associated Code Exercise below.

TIP: When writing threaded network classes, it is easiest to write the logic of your `Run()` member function of your Thread class first. Due to the convenient blocking, suspending and resuming of

threads, you can easily implement the structure of your entire protocol in your `Run()` member function.

Connectionless Datagram Communications

So far we've mainly discussed TCP. However, the PowerPlant network classes also support UDP, User Datagram Protocol. Whereas TCP offers a reliable, full-duplex, connection-oriented stream service, UDP offers best-effort, connectionless datagram delivery with an optional checksum. UDP still allows you to specify a port on the remote computer, however, which differentiates it from the lower-level IP.

UDP is extremely easy to use. In fact, it is very similar to TCP except for the fact that you do not call `Connect()` and `Disconnect()`. Once you bind to a local port, the functions you need to make use of are `SendPacketData()` and `ReceiveFrom()`. By calling these functions you can send and receive datagrams (small packets of data that stand on their own) between two remote computers. Protocols that simply return the local time, or a short text message such as a quote, lend themselves to UDP quite nicely.

NOTE: Using UDP is simpler than TCP but don't use it unless you need to. Some simple protocols support UDP for convenience, but UDP is not reliable. TCP has built-in functionality to ensure reliable delivery of data, quickly, in both directions. UDP communications are prone to errors if the connection is not pristine in quality. Only experienced network programmers should use UDP.

Summary of Networking in PowerPlant

Communications and networking between applications is becoming increasingly important in today's diverse computer culture. TCP/IP is an important protocol for applications to implement because it is cross-platform in nature. A TCP/IP application on a Macintosh can

communicate easily with a TCP/IP application on a PC running Windows, a UNIX or NeXT workstation, a BeBox or a mainframe.

There are numerous standard protocols today that you can implement using the PowerPlant network classes including HTTP, FTP, and SMTP. You can also define your own protocols built upon the TCP/IP implementation to extend your application and make it "Internet-savvy." The ability to access remote computers and make use of their resources makes your application that much more powerful.

Code Exercise for Networking

The SimpleClient and SimpleServer applications show how to implement both a client and server application using the PowerPlant network classes. Although there are numerous ways to use the classes provided here, these examples should give you a simple introduction to one implementation. In this section we cover two code exercises:

- SimpleClient
- SimpleServer

SimpleClient

In the first part of this exercise you implement portions of the SimpleClient application.

The purpose of this exercise is to give you experience using networking in PowerPlant—the functions you override and the tasks you perform. This exercise is not intended as a tutorial on general networking techniques. For more information on networking in general you should consult a text on Internet protocols and computer networking. A short list was mentioned earlier in this chapter.

SimpleClient is just that, a simple client. It implements a Telnet-like terminal that allows you to type characters which are sent to the remote computer and are echoed back to the terminal and displayed. Although you can connect to most any server, the SimpleServer application (discussed below) is designed specifically to be used with SimpleClient.

Let's take a look at the important steps needed to implement client-side networking in your application. In the first part of this exercise you write the code to:

- Create a TCP endpoint for the client.
- Bind to the local endpoint.
- Open an outgoing connection.
- Send data to the remote computer.
- Receive data from the remote computer.
- Disconnect from the remote computer.
- Unbind the TCP endpoint.
- Quit your application. Cleaning up when you're through.

1. Create a TCP endpoint for the client.

`StartSession()` `CClientConnection.cp`

Before you can communicate via TCP you must create a TCP endpoint. You use the `UNetworkFactory::CreateTCPEndpoint()` function to do this. As usual, in these exercises existing code is in italics.

```
mTCPEndpoint =
UNetworkFactory::CreateTCPEndpoint();
mTCPEndpoint->QueueSends();
mTCPClientThread = new
CTCPClientThread(mTCPEndpoint, mTerminalPane,
this);
mTCPClientThread->Resume();
```

Note that we also enable the "Queue Sends" mechanism at this time. Because this is an option for the endpoint, this is the perfect time to enable it.

You should also note that the existing code that follows creates a thread for our endpoint and immediately "kick-starts" the thread by calling the `Resume()` function which begins execution of the thread's `Run()` method. The `Run()` method is the heart of our client implementation.

2. Bind to the local endpoint.

`Run()` `CTCPClientThread.cp`

After you have created your TCP endpoint, you must bind to it. Binding "connects" you to the endpoint and allows you to properly communicate through it. Because we are initiating the communications we don't care which local port we communicate through, therefore we create an LInternetAddress object passing 0 for both the host address and host port. Once instantiated we pass the address to the endpoint's `Bind()` method.

```
LInternetAddress address(0, 0);
mEndpoint->Bind(address);
```

Because we are using threads, the `Bind()` method will block the thread until the bind is complete. When it is complete the thread will be resumed automatically and will continue to execute.

Note that if the bind fails for any reason, our try/catch block will catch the exception and essentially abort the `Run()` method.

3. Open an outgoing connection.

`Run()` `CTCPClientThread.cp`

Once we have the port in our control we need to actually open an outgoing connection using the port. This is a very simple task. First we create an LInternetAddress based on the address that the user entered when they initiated the connection. This might be in the form of a dotted-decimal IP number (127.0.0.1) or a name (www.metrowerks.com). Once we have the LInternetAddress object created we can simply pass it to the `Connect()` method of the endpoint. The `Connect()` method will automatically block our thread, perform any needed DNS lookups, connect to the remote computer and the resume our thread.

```
LInternetAddress* remoteAddress = mClientMaster-
>GetRemoteAddress();
mEndpoint->Connect(*remoteAddress);
```

Note that if the connect fails for any reason, our try/catch block will catch the exception and essentially abort the `Run()` method.

4. Send data to the remote computer.

`InternalSend()` `CClientConnection.cp`

Now that we have a connection open between our client and a server on the remote computer, we can easily send and receive data between the two processes. This is as simple as passing the data to any of the numerous "send" functions supported by the endpoint. In this case we use the `Send()` method passing in a pointer to the data and the length of the data.

```
mTCPEndpoint->Send(theData, theLength);
```

Because we enabled the queue sends mechanism when we created the endpoint, the `Send()` method will return immediately allowing our program to continue with no delay. It will then send the data "behind the scenes" as we continue other processing.

5. Receive data from the remote computer.

`Run()` `CTCPClientThread.cp`

In order to receive data from the remote computer we first must know what type of data we will be receiving. In this case we know that the data is sent a character at a time and is echoed back to us in the same format. Therefore, we can easily call the endpoint's `ReceiveChar()` method to receive a character of data at a time. In our `Run()` method we simply loop, receiving characters, until we either initiate a disconnect or the remote endpoint does so.

```
char theChar;
mEndpoint->ReceiveChar(theChar, 5);
mTerminalPane->DoWriteChar(theChar);
```

The ReceiveChar() method is called passing a buffer to store the received character in as well as a timeout value. In this particular case, if no characters are received within 5 seconds, the function will return and our loop will continue. You will also note that once received, we immediately write the character to our terminal pane so the user can see it has been echoed.

6. Disconnect from the remote computer.

`Run()` `CTCPClientThread.cp`

When we are ready to disconnect from the remote computer we simply call the `Disconnect()` method of the endpoint. Our thread

will block, the disconnect will take place, and our thread will resume.

```
mEndpoint->Disconnect();
```

7. Unbind the TCP endpoint.

`Run()` `CTCPClientThread.cp`

Once the `Disconnect()` method resumes our thread we continue the process by unbinding from the port. This is as simple as calling the `Unbind()` method of the endpoint.

```
mEndpoint->Unbind();
```

One thing to note is that you don't have to unbind at this point. If you were going to open another connection immediately you might choose to use the same port. In this case you may forego the unbind and simply open a connection to another remote computer instead.

8. Quit your application.

`main()` `CSimpleClientApp.cp`

When your application quits, the most important thing you have to do is call `LCleanupTask::CleanUpAtExit()`. This function ensures that all tasks are cleaned up and are properly destroyed before the application quits. Not calling this function can cause a crash. Even if you don't crash, other TCP/IP applications may not function properly until you restart your computer.

```
LCleanupTask::CleanUpAtExit();
```

9. Build and run the application.

When the application builds successfully and runs, select `New Session...` from the File menu and enter in the name and port number of the remote computer you wish to connect to. Once entered press the OK button.

Once connected you can type text into the terminal and it will echo back to you. To disconnect simply close the terminal window.

Congratulations! You've implemented a network-savvy application using the PowerPlant network classes.

SimpleServer

In the second part of this exercise you implement portions of the SimpleServer application.

SimpleServer is just that, a simple server. It implements a Telnet-like server that allows you to send characters to it which are immediately echoed back to the sender.

Let's take a look at the important steps needed to implement server-side networking in your application. Much of what you see here will be similar to what you implemented on the client side except for a few important differences which we outline below. In the second part of this exercise you write the code to:

- Create a TCP endpoint for the server.
- Bind to the local endpoint. So your server can "listen" for a connection.
- Listen for an incoming connection request.
- Accept an incoming connection request.
- Handle the connection.

1. Create a TCP endpoint for the server.

`WaitForConnections()` `CSimpleTCPServer.cp`

Before you can communicate via TCP you must create a TCP endpoint. You use the `UNetworkFactory::CreateTCPEndpoint()` function to do this. As usual, in these exercises existing code is in italics.

```
mEndpoint =
UNetworkFactory::CreateTCPEndpoint();
mEndpoint->AddListener(this);
```

Note that we also add the CSimpleTCPServer object as a Listener to the endpoint. This allows us to receive the `T_LISTEN` message from the endpoint when a remote connection request is received. This makes use of the standard Broadcaster/Listener relationship used throughout the PowerPlant framework.

2. Bind to the local endpoint.

`Run()` `CTCPServerThread.cp`

Binding to the TCP endpoint is performed much the same as the client side of the connection with two notable exceptions. The first being that we must specify a local port number to bind to. This is so the client application knows how to contact our server. Remember, when you connect to a remote computer using TCP you not only supply the address of the computer but also the port number on that computer. When we actually call the `Bind()` method of the endpoint we also pass in the maximum number of connections we would like to listen for. This allows TCP to pass multiple connection requests to our server simultaneously instead of simply turning them away.

```
LInternetAddress address(0, mPort);
mEndpoint->Bind(address, mMaxConnections);
mServerMaster->BindCompleted();
```

You will also note that after the bind has completed (our thread is blocked by the bind then resumed) we call a function of the master server object called `BindCompleted()`. This is simply a mechanism to pass a message back to the object that handles our user interface to let it know that the bind has completed successfully and it can safely display our server's window.

3. Listen for an incoming connection request.

`Run()` `CTCPServerThread.cp`

There is actually nothing to do to begin our server listening for incoming connection requests. Because we passed in a value to the `Bind()` method specifying how many listeners we can handle, we are automatically listening once the bind is complete. We do however choose to suspend our thread until an actual connection request is received.

```
Suspend();
```

When a connection request is received, you will remember that the endpoint is set up to broadcast a `T_LISTEN` message to the CSimpleTCPServer object. When this message is received by the CSimpleTCPServer object it simply resumes the server thread. See `CSimpleTCPServer::ListenToMessage()`.

4. Accept an incoming connection request.

`Run()` `CTCPServerThread.cp`

Once our thread is resumed (by the reception of the `T_LISTEN` message) we simply call the Listen() method of the server endpoint.

```
mEndpoint->Listen();
```

After this point we can easily create a new TCP endpoint, known as the responder endpoint (see `CTCPResponder::Accept()`), create a new responder thread (see `CTCPResponderThread::Run()`) and within it call the `AcceptIncoming()` method of the server endpoint, passing our responder endpoint as the only parameter. This effectively hands off the connection currently serviced by the server endpoint to the responder endpoint and opens the connection fully.

5. Handle the connection.

From this point onward you can perform the same tasks in the responder as you did in the client. This includes sending and receiving data, disconnecting, unbinding and quitting your application. The responder, in many cases, can even descend from the same base class as your client. What could be easier?

As extra credit, walk yourself through the code exercises again but now looking for the UDP specific code as opposed to the TCP specific code that we outlined above.

This program can be extended in many exciting ways. You could easily send complex data across the connection, not just single characters. For example, you might have a picture display. You could send a picture pasted into the display to the remote computer.

You might also consider using the Sound Manager to send live audio across the connection. How about MIDI data? You could even send Event Manager events to the remote computer. Whenever you click the mouse in a window on your computer it causes a click to occur on the remote computer! This could be the beginnings of a collaborative drawing application. And how about Apple Events across the Internet?

The possibilities are limitless. Good luck, and happy networking!

8

Internet Programming in PowerPlant

This chapter discusses how to use the PowerPlant Internet classes to create a variety of Internet-enabled programs.

Introduction to Internet Programming in PowerPlant

The global collection of networks known as the Internet has grown in size at a staggering rate. Businesses, schools, governments, and individuals are making use of the interconnected nature of the Internet to conduct their daily affairs. Many tools help in these endeavors: electronic mail, the World Wide Web, and file transfer being but a few.

The primary networking protocol stack used on the Internet is the collection of protocols known as Transmission Control Protocol/Internet Protocol (TCP/IP). On top of this foundation exists a large collection of mostly standardized, task-specific protocols. These protocols are specified in the Request For Comment (RFC) documents which are the definitive source for implementation details.

Several of these task-specific protocols have been implemented in the PowerPlant library of classes. Electronic mail is represented with classes covering the Simple Mail Transport Protocol (SMTP) for sending messages to mail servers, and the Post Office Protocol version 3 (POP3) for retrieving messages from mail servers. The main protocol of the World Wide Web, the Hierarchical Text Transfer Protocol (HTTP), is provided. File transfer is included with the File

Transfer Protocol (FTP) classes. Finally, a collection of helper classes exist to make preparing and interpreting data sent by these protocols simpler to code.

The PowerPlant implementations of these classes are built on top of the PowerPlant network classes documented elsewhere in this manual. By using the network classes as the foundation, the Internet classes work well with either MacTCP or Open Transport installed on the target Macintosh.

WARNING! The Internet classes make use of the threaded version of the PowerPlant network classes. You will need to check that the Thread Manager is installed on the host computer for your application to work properly.

This chapter's topics include:

- Internet Programming Strategy—PowerPlant's approach to using Internet Protocols
- Internet Classes—a detailed examination of the PowerPlant implementation of some Internet protocols
- Implementing an Internet Enabled Application—how to add basic Internet protocols to your application
- Summary of Internet Protocol Usage in PowerPlant
- Code Exercise

Where to Learn More About Internet Protocols

The Internet is always changing, and even the protocols that are described in this chapter are constantly evolving. This chapter shows you how to make use of the protocol classes, but does not teach you the intricacies and nuances of the protocols themselves. For that, we recommend that you review the ultimate resource for the protocols, the Request For Comment (RFC) documents. RFCs describe in great depth how protocols function, and they explain some of the implementation details that are useful to understand when writing Internet software.

This chapter's descriptions of the PowerPlant Internet classes and their application assumes that you have a basic familiarity with Internet protocols and creating networking software. You should also understand the fundamentals of using Mac OS Threads and the PowerPlant classes that support them. There are many issues that you must consider when implementing a robust communications program, and experience is the best guide. There are many other books and resources that you can read to learn about Internet protocols and programming.

Comer, Douglas E. *Internetworking with TCP/IP, Volume 1, Principles, Protocols, and Architecture*. Prentice Hall.

Comer, Douglas E. *Internetworking with TCP/IP, Volume 3, Client-Server Programming and Applications.* Prentice Hall.

Stevens, W. Richard. *TCP/IP Illustrated, Volume 1-3.* Addison Wesley.

Internet Engineering Task Force at http://www.ietf.cnri.reston.va.us/ (for RFCs)

Software Requirements

The PowerPlant Internet classes require several pieces of system software. MacTCP (also known as "classic networking") or Open Transport must be installed and properly configured for the Internet classes to operate. If you want to be able to connect with other machines, the computer on which your application runs must also be connected to a network that supports TCP/IP. The Internet classes will work just fine regardless of whether you are connected via Ethernet, a serial connection using Point to Point Protocol (PPP), or some other physical connection. However, you may want to adjust the operation of your software depending on the performance of the link.

Additionally, the Thread Manager must be installed if it is not already part of the version of the Mac OS running on the target computer.

Internet Programming Strategy

The PowerPlant Internet classes implement a general foundation that easily supports Internet protocols modeled on a command and response scheme. On top of this foundation is support for some of the most common Internet standard protocols. The different protocol APIs provide a simple interface that allows you to concentrate on the operation you wish to complete, not the underlying details.

TIP: Depending on your programming situation, you may want to use the Internet classes apart from the rest of PowerPlant. The Internet classes currently require the following other components of the PowerPlant library: the Internet Class hierarchy, the Network hierarchy, the Thread classes, LPeriodical, LArray, UMemoryMgr utilities, LBroadcaster, LListener, and the ANSI library.

This section discusses the following topics:

- Generic Internet Protocol Interface—PowerPlant's abstract foundation for representing Internet protocols
- Specific Internet Protocol Interfaces—PowerPlant's implementation of popular Internet protocols
- Internet Messages—the general representation of data sent by many Internet protocols
- General Utilities—PowerPlant's tools for making Internet programming easier
- Strategic Summary

Generic Internet Protocol Interface

Many Internet protocols follow the same general pattern in how they communicate between two computers. The connection is opened by the sending computer, the receiver acknowledges the connection, then the sender starts writing command sequences with optional data one at a time. The receiver accepts each command and replies with a response code and optional data.

The Internet classes implement this generic behavior with two base classes:

- LInternetProtocol
- LInternetResponse

LInternetProtocol is used to create the connection between your program and the destination program on a remote computer. This base class provides the infrastructure for protocols that follow the command and response model.

LInternetResponse embodies the typical format of a response that you will receive from the remote program when it acknowledges your program's commands.

Specific Internet Protocol Interfaces

The PowerPlant Internet classes implement several of the more popular Internet protocols in use today. These classes all follow the general command and response model of communication. The classes currently implemented include:

- LSMTPConnection
- LSMTPResponse
- LPOP3Connection
- LPOP3Response
- LHTTPConnection
- LHTTPResponse
- LFTPConnection
- LFTPResponse

LSMTPConnection and LSMTPResponse implement the Simple Mail Transport Protocol (SMTP) which is the primary transmission protocol for electronic mail messages. The classes implement the basic command set of the SMTP specification (RFC822).

LPOP3Connection and LPOP3Response implement the Post Office Protocol version 3 (POP3). This set of classes can be used to retrieve mail from a compliant mail server. The classes implement some of

the optional POP3 commands such as TOP and APOP, an alternate authentication method (RFC1725).

LHTTPConnection and LHTTPResponse implement the Hierarchical Text Transport Protocol (HTTP). HTTP is used most frequently for communication with World Wide Web (WWW) servers. These classes give you all of the necessary tools for using the basic features of HTTP version 1 including the GET, POST, and HEAD methods (IETF draft 4—see RFC web site).

LFTPConnection and LFTPResponse implement the File Transfer Protocol. These classes allow you to send, retrieve, and manipulate files as described in the FTP specification (RFC959).

NOTE: This chapter does not describe the inner workings of the various Internet protocols supported by the PowerPlant Internet class library. The ultimate source of information for these protocols can be found in their specifications in the Request For Comments documents which can be found at the address given earlier in this chapter.

Internet Messages

Some Internet protocols, such as SMTP and HTTP, require that data be sent in a special format (RFC822). These encapsulated messages are composed of two parts:

1. a collection of headers which provide protocol specific information regarding the handling of the data, and
2. a message body that contains your data (perhaps encoded in a certain way).

The PowerPlant Internet classes support this concept with a collection of classes that implement messages:

- LInternetMessage
  - LMailMessage
  - LHTTPMessage

- LHeaderField
- LHeaderFieldList
- LMailMessageList

LInternetMessage is a base class that implements the basic behavior of an RFC822 message. This class provides the ability to easily manipulate the headers and body of a message.

LMailMessage is a message for electronic mail and is used by both SMTP and POP3. It provides a simple interface for accessing mail specific header fields. The class supports simple MIME enhancements including multipart messages.

NOTE: MIME stands for Multipurpose Internet Mail Extensions (RFC1521) and is a specification for a set of message headers that describe a variety of enhanced message body types and content. MIME messages can include special encodings, multiple parts, binary information, etc.. HTTP 1.0 is not fully MIME compliant. HTTP uses the MIME headers to determine the content type of the message, but only supports a small subset of the available types at this time.

LHTTPMessage is a message specifically formatted for HTTP. It provides a simple interface to access header fields unique to the protocol. The class implements the minimal MIME support required by the HTTP specification.

LHeaderField is a utility class used by the message classes to store and construct RFC822 style header fields.

LHeaderFieldList is another utility class that maintains an array of header fields during the construction of a message.

LMailMessageList is a utility class that maintains an array of pointers to LMailMessage objects.

General Utilities

The PowerPlant Internet classes make use of a number of general utility functions and classes:

- LDynamicBuffer
- UInternet
- MD5
- UUEncode

LDynamicBuffer is used throughout the Internet classes to provide storage buffers that can grow or shrink whenever you change their contents.

UInternet contains a number of useful utility functions that includes routines to encode data for a variety of different protocols.

MD5 is a collection of C functions that implement the MD5 message digest encryption algorithm.

UUEncode is a collection of C functions that implement the uuencode character encoding algorithm (based on RFC1113).

NOTE: The details of the different encoding routines are beyond the scope of this chapter. You should refer to a standard text on character encoding and encryption for more details.

Strategic Summary

Depending on which protocol you choose to use in your program, there are really two levels of detail you can explore within the PowerPlant Internet classes:

- Use the simple wrapper function in those protocols that support it to encapsulate your task into one function call
- Use the detailed protocol functions to manage each step of the transaction

The first choice gives you "fire and forget" functionality for the most common tasks in each protocol. You should use this option when you want to add basic Internet functionality to your program and don't need fine control over the connection.

The second choice is more appropriate if you need a high degree of control over the flow of the session. One example might be when you are implementing a full client application for your chosen protocol. In this kind of program, you want to manage the transaction each step of the way. Adding an Internet protocol to your program is still a relatively simple procedure. You open a connection to the desired remote computer, alternate between sending commands with optional data and receiving their responses, and finally closing down the connection.

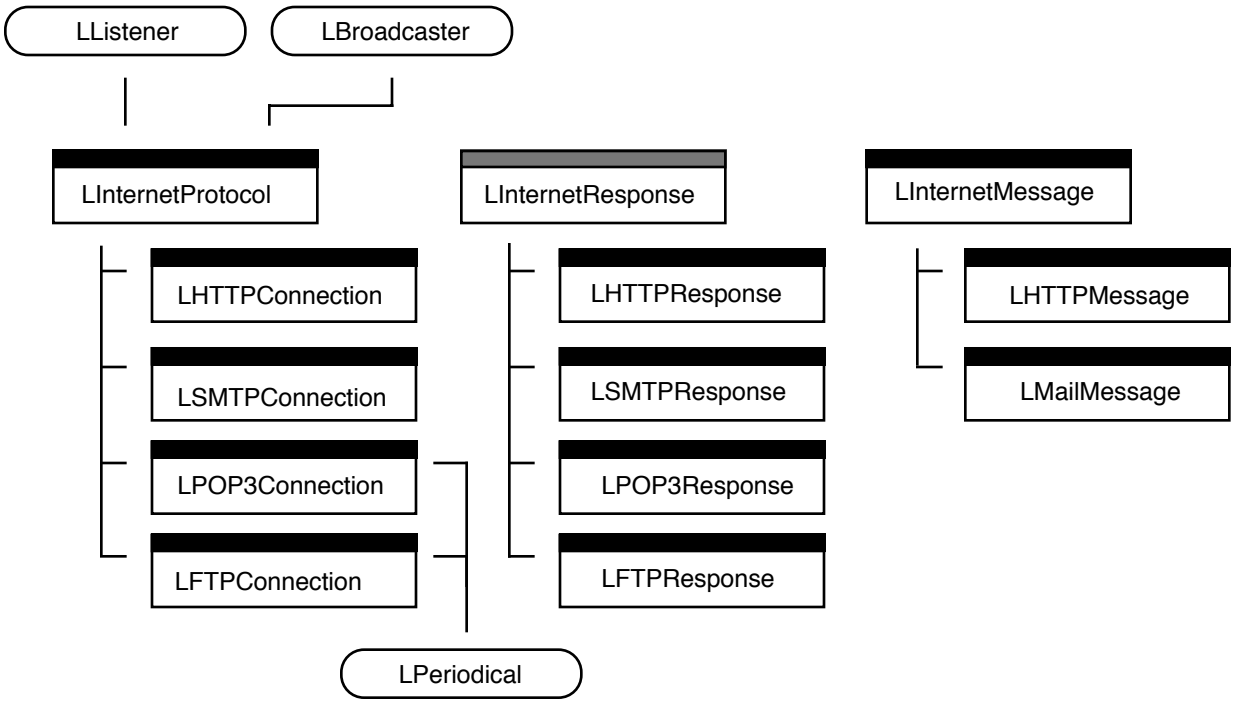
Internet Classes

Before reading this section, you should have a basic understanding of the operation of the PowerPlant Internet class library. You should read "Internet Programming Strategy" on page 256 to review the general concepts of the library. In this section, we will take a closer look at the primary classes, and describe their more important functions and behaviors. Figure 8.1 is an illustration of the classes that we will cover and their relationship to one another.

The gray bar indicates an abstract class.

NOTE: These classes were designed so that you can make use of them with a minimal amount of PowerPlant. See "Internet Programming Strategy" on page 256 for a list of dependencies if you wish to use the Internet classes in non-PowerPlant code.

Figure 8.1 The primary Internet classes



This section discusses the following classes:

- LInternetProtocol
  - LSMTPConnection
  - LPOP3Connection
  - LHTTPConnection
  - LFTPConnection
- LInternetResponse
  - LSMTPResponse
  - LPOP3Response
  - LHTTPResponse
  - LFTPResponse
- LInternetMessage
  - LMailMessage

  - LHTTPMessage
- LDynamicBuffer
- LHeaderField
- Other Classes

LInternetProtocol

LInternetProtocol is the foundation on which PowerPlant's Internet functionality is based. It is the base class for implementing those protocols that follow a command and response model of communication. It is a subclass of LListener so that it may capture networking events generated from its LTCPEndpoint object. LInternetProtocol also inherits from LBroadcaster and uses this functionality to provide a progress notification mechanism for clients of the protocol object.

LInternetProtocol is itself relatively simple, providing the basic tools to implement specific protocols. This description outlines the primary member functions you are likely to use in your code. For more details, refer to the PowerPlant source code.

LInternetProtocol's data members are all protected from direct access, you should make use of the provided accessor functions if you need to get or change values.

NOTE: By making use of accessor functions to manipulate shielded data, you protect your code from being compromised by future changes in the underlying PowerPlant Internet class architecture.

Table 8.1 Important LInternetProtocol functions

Function	Purpose
`LInternetProtocol()`	constructor, you must supply a reference to the thread in which the protocol is being implemented
`Connection Management`	
`Connect()`	manually opens a connection to remote system addressed via a DNS format address
`Disconnect()`	manually closes the connection to remote system
`Abort()`	stops the network operation
`Data Transfer`	
`SendData()`	sends data buffer in SendSize chunks, reporting progress periodically
`Data Buffer Management`	
`SetSendSize()`	sets the size chunks SendData() will break a buffer into for transmission
`GetSendSize()`	returns chunk size
`Thread Management`	
`SetThread()`	sets the thread to yield to on data arrival
`Progress Notification`	
`BroadcastProgress()`	sends a coded message and SProgressMessage buffer to any LListeners linked to the protocol object

Function	Purpose
`SetMinBroadcast-Ticks()`	sets the frequency of progress notification in ticks
`GetMinBroadcast-Ticks()`	gets the progress notification in ticks

LInternetProtocol is a base class that handles all of the details of the network connection for you. Perhaps its most helpful trait is that it hides (and handles) interaction with the lower level network classes. It creates an LTCPEndpoint, handles addressing, binding, connections, data transfer, and other minutia.

LInternetProtocol doesn't implement a specific protocol, but rather provides the basic tools with which HTTP, SMTP, POP3, and other command and response protocols can be constructed. You will rarely need to instantiate a plain LInternetProtocol object.

The protocols written on top of LInternetProtocol have many convenience functions that are particular to the individual protocols. You will most frequently make use of the convenience functions and not need to worry about the functions implemented at this level. One specific example is the connection management functions. The wrapper functions are simple interfaces in the protocols detailed below that handle the opening and closing of connections for you.

An important aspect of this class is that LInternetProtocol and its derivatives are intended to be created from within an LThread object. The standard procedure is to create a thread for each instance of the protocol (often for each separate connection), and within the thread's body, create the protocol object. See the code exercises below for an example.

NOTE: The Internet classes make heavy use of threading. You should be familiar with Mac OS threads in general, and the PowerPlant LThread class hierarchy in particular. See the chapter on Threads for more information about using threads in your projects.

A typical scenario for using a connection can be summarized like this:

- Collect the connection information (DNS address and port of the remote computer).
- Create a thread that will drive the connection.
- Create the connection object within the thread, and pass a reference to the thread to the connection's constructor.
- Set the thread to be a listener to the connection object which will allow the thread to capture progress messages broadcast from the connection.
- Send any data you may have on the connection, and possibly receive progress reports.
- When the transaction is done, let the thread run down and clean up the objects previously created.

LSMTPConnection

LSMTPConnection implements the Simple Mail Transport Protocol (SMTP). Internet clients usually use SMTP to send electronic mail messages to SMTP aware servers. The current PowerPlant implementation of SMTP is designed with this behavior in mind. LSMTPConnection is a very simple class, providing member functions that wrap the process of sending mail messages into one function call.

Table 8.2 Important LSMTPConnection functions

Function	Purpose
`LSMTPConnection()`	constructor
`Connection Management`	
`Connect()`	defaults to port 25
`Protocol Commands`	

Function	Purpose
`SendOneMessage()`	wrapper functions that sends one LMailMessage object, defaults to port 25
`SendMessages()`	wrapper function that sends one or more LMailMessages provided in an LMailMessageList array object, defaults to port 25

The protocol command functions `SendMessages()` and `SendOneMessage()` handle the opening and closing of a connection. You will rarely need to use the `Connect()` function.

SMTP defaults to TCP port 25, and if you do not specify otherwise, LSMTPConnection will use the standard port number. LSMTPConnection implements the basic SMTP protocol as defined in RFC821.

NOTE: SMTP has been extended in later RFCs to provide more advanced and efficient data handling (SMTP Service Extension--RFC1651). LSMTPConnection does not currently support these extensions.

LPOP3Connection

LPOP3Connection implements the Post Office Protocol, version 3 (POP3). POP3 is used to retrieve electronic mail messages from a compliant mail server. It offers authenticated access to a remote mail box and has the ability to retrieve all or portions of a mail drop.

When implementing a mail retrieval client, you will often want more functionality than simply gathering mail messages from a remote host. You may want to retrieve some messages and leave others on the server. You may want to provide your user with housekeeping functionality, such as the ability to just check mail message headers, or to delete mail without downloading messages first. LPOP3Connection provides both simple to use wrapper functions

for retrieving mail and functions for more detailed access to the POP3 command set.

LPOP3Connection multiply inherits from both LInternetProtocol and LPeriodical.

Table 8.3 Important LPOP3Connection functions

Function	Purpose
`LPOP3Connection()`	constructor
`Connection Management`	
`Connect()`	makes a connection with supplied authentication credentials, defaults to port 110
`SpendTime()`	periodically calls `NoopServer()` to maintain connection
`Protocol Commands`	
`GetOneMessage()`	wrapper function for connecting, authenticating, and collecting one LMailMessage from the designated mail box and identified by a message number, defaults to port 110
`GetMessages()`	wrapper function for connecting, authenticating, and collecting all messages in the designated mail box and returning them in an LMailMessageList array of LMailMessages, defaults to port 110
`GetHeaders()`	wrapper function for connecting, authenticating, and collecting all message headers in the designated mail box and returning them in an LMailMessageList array of LMailMessages, defaults to port 110

Function	Purpose
`CollectAllMessages()`	gets all messages from the open connection and places them in an LMailMessageList array of LMailMessages
`CollectAllHeaders()`	gets all headers from the open connection and places them in an LMailMessageList array of LMailMessages
`GetMailMessage()`	retrieves one message identified by its session mail box message number
`GetTop()`	retrieves one header identified by its session mail box message number
`DoList()`	retrieves an LList of POP3ListElem structures culled from the results of the LIST command
`DoUIDL()`	retrieves an LList of POP3ListElem structures culled from the results of the UIDL command
`DeleteMessage()`	removes the message identified by its session mail box message number from the remote computer
`NoopServer()`	sends a NOOP command to the remote computer
`ResetServer()`	sends a RSET command to the remote computer
`ServerStatus()`	retrieves the mail box message count and size from the server
`SendQUIT()`	sends a QUIT command to the remote computer

The protocol command functions `GetOneMessage()`, `GetMessages()`, and `GetHeaders()` are wrapper functions that handle the

opening and closing of a connection. If the connection is already open, they will not automatically close it. Likewise, if the connection is closed, the connection will be closed on completion of the function.

The lower level protocol command functions require that the connection be open and the user be authenticated. If this is not true, the functions will throw exceptions.

Many of the LPOP3Connection member functions require a message number as one of their parameters. As described in the POP3 specification, message numbers do not uniquely identify a mail message across different connections to the mail server. Only rely on a number during a single session. If you need a unique identifier for a message, try to use the UIDL results to match the unique id to the current message number. See the RFC for details.

POP3 defaults to TCP port 110, and all connection related functions will use the standard port unless you supply another port number. LPOP3Connection implements the standard POP3 version as described in RFC1725. It includes implementations of the optional commands UIDL, TOP, and APOP. Most, but not all, POP3 servers support this optional command set. Your code should prepare for the worst. Be especially careful when using the header related functions—they use the TOP command, which is often not implemented on older POP3 servers—and be sure to trap any exceptions.

WARNING! POP3 connections are often open through a series of command and response exchanges. Some servers will close the connection to your program if they reach some arbitrary timeout period. To counter this, LPOP3Connection also inherits from LPeriodical. When the `SpendTime()` member function reaches a designated delay period, it tickles the server by sending a NOOP POP3 command. If you use the Internet classes outside of the normal PowerPlant framework, you will have to account for this functionality by periodically calling the `SpendTime()` function yourself.

LHTTPConnection

LHTTPConnection implements the Hierarchical Text Transfer Protocol (HTTP) for connections to software such as Web server programs. HTTP is a stateless protocol, it requests and potentially receives its data in a single transaction with the remote computer. The connection is opened and closed between each transaction. Usually the remote system does not maintain information about your program between these transactions (hence, the statelessness).

HTTP offers a variety of commands, called "methods" in the HTTP specification. LHTTPConnection supports three of the more popular commands: GET for retrieving a resource, HEAD for retrieving information about a resource, and POST for sending information.

Table 8.4 Important LHTTPConnection functions

Function	Purpose
`LHTTPConnection()`	constructor
`Connection Management`	
`Connect()`	defaults to port 80
`Protocol Commands`	
`RequestResource()`	one transaction wrapper for sending an HTTP command and URL specified resource, defaults to a GET on port 80
`Get()`	one transaction wrapper for a GET on the URL specified resource, defaults to port 80
`Head()`	one transaction wrapper for a HEAD on the URL specified resource, defaults to port 80
`Post()`	one transaction wrapper for a POST on the URL specified resource, defaults to port 80

Function	Purpose
`Protocol Data Accessors`	
`SetCheckContentLength()`	set true (default) to have LHTTPConnection verify that the data returned in the LHTTPMessage is the same length as the Content-Length header field indicates
`GetCheckContentLength()`	returns true if content length checking is enabled

In most cases, you will want to allow the different functions that implement the protocol commands (e.g. `RequestResource()`, etc.) to handle the opening and closing of the connection. It is relatively rare that you will need to use the `Connect()` and `Disconnect()` member functions.

HTTP defaults to TCP port 80, and if you do not specify otherwise, LHTTPConnection will use the standard port number. LHTTPConnection implements HTTP version 1.0.

NOTE: Uniform Resource Locators (URLs) are a way by which any item on the Internet can be specified and found. Think of URLs as your complete home address, by giving someone your URL they can locate and go directly to your resource. See RFC1738 for a detailed explanation of the URL format and how they can be used.

LFTPConnection

LFTPConnection implements the File Transfer Protocol (FTP). FTP offers authenticated access to a remote file system and it permits a variety of file and directory manipulation operations. FTP is most frequently used to send and retrieve files between computers.

FTP is a full featured protocol containing many different commands to manipulate file system objects. FTP was designed to abstract

away the differences between file systems on different operating systems. When implementing an FTP client, you may want to have detailed control over the FTP session. Other times you may want to simply retrieve or send a single file. LFTPConnection implements easy to use wrapper functions for the simplest of cases while also providing a rich set of functions for detailed manipulation of the protocol.

LFTPConnection multiply inherits from both LInternetProtocol and LPeriodical.

Table 8.5 Important LFTPConnection functions

Function	Purpose
`LFTPConnection()`	constructor
`Connection Management`	
`Connect()`	defaults to port 21
`SpendTime()`	periodically calls `NoopServer()` to maintain connection
`Protocol Commands`	
`PutFile()`	wrapper function for connecting, authenticating, and sending one file to a remote computer, defaults to port 21
`GetFile()`	wrapper function for connecting, authenticating, and retrieving one file from a remote computer, defaults to port 21
`RenameRemoteFile()`	convenience function for renaming remote files
`ListFolder()`	retrieves an LDynamicBuffer containing the full or name-only directory listing of the target directory

Function	Purpose
`SendRETR()`	sends a RETR command to get a file from remote computer
`SendPASV()`	sends a PASV command to specify a non-default port for the remote computer to listen to for data connections, default is for remote computer to initiate data connection
`SendSTOR()`	sends a STOR command to send a file to remote computer
`SendPORT()`	sends a PORT command to specify non-default client address and port to use by data connection
`SendTYPE()`	sends a TYPE command to specify the representation of the data to be transmitted, default is ASCII
`SendChangeDir()`	sends a CWD command to set a new working directory on the remote computer
`SendChangeDirUp()`	sends a CDUP command to set the working directory to the parent of the current working directory
`SendGetWorkingDir()`	sends a PWD command to retrieve the current working directory (in an LFTPResponse object)
`SendDeleteRemote-Dir()`	sends a RMD command to remove the named directory
`SendCreateRemote-Dir()`	sends a MKD command to make a specified directory
`SendSystemRequest()`	sends a SYST command to retrieve the type of the remote operating system (in an LFTPResponse object)

Function	Purpose
`SendDelete()`	sends a DELE command to remove the named file
`SendRenameFileFrom()`	sends a RNFR command to specify a file to rename (must be immediately followed by a call to `SendRenameFileTo()`)
`SendRenameFileTo()`	sends a RNTO command to specify a new file name for a previously identified file (must call `SendRenameFileFrom()` immediately before this function)
`NoopServer()`	sends a NOOP command to the remote computer
`SendQUIT()`	sends a QUIT command to remote computer
`SendLIST()`	sends either LIST or NLST to retrieve the directory listing of the current working directory
`Protocol Data Accessors`	
`GetLastResponse()`	returns a reference to an LFTPResponse used by the last command

The protocol command functions `GetFile()` and `PutFile()` are wrapper functions that handle the opening and closing of a connection. If the connection is already open, they will not automatically close it. Likewise, if the connection is closed, the connection will be closed on completion of the function.

The lower level protocol command functions require that the connection be open and the user be authenticated. If this is not true, the functions will throw exceptions.

Unlike other protocols, FTP makes use of multiple connections between the client and server computers. A control connection is

maintained throughout the lifetime of the FTP session and is used to send commands and receive their status responses. FTP uses TCP port 21 as the default command connection port for all commands unless you specify otherwise. A separate connection—called the data connection—is created when file system objects are transmitted to or from the client. Data connections use a separate TCP port for their connections.

LFTPConnection provides an abstraction of the entire FTP session. The LFTPConnection object maintains the command and response connection. An internal class, LFTPDataConnection, is used by LFTPConnection to manage the data portion of a command transparently.

LFTPConnection implements the basic subset of commands found in RFC959.

WARNING! FTP connections are usually open through a series of command and response exchanges. Some servers will close the connection to your program if they reach some arbitrary time-out period. To counter this, LFTPConnection also inherits from LPeriodical. When the `SpendTime()` member function reaches a designated delay period, it tickles the server by sending a NOOP FTP command. If you use the Internet classes outside of the normal PowerPlant framework, you will have to account for this functionality by periodically calling the `SpendTime()` function yourself.

LInternetResponse

Internet protocols that follow the command and response model tend to have a similar scheme for formatting their response data. These responses are usually ASCII strings which start with a code (often a numeric code), followed by a separator (a space), then optional text that is often used to supply human readable equivalents to the initial code. Most protocols terminate the response with a standard delimiter, usually a carriage-return and line-feed (CRLF).

LInternetResponse is a simple class that encapsulates the response concept.

Table 8.6 Important LInternetResponse functions

Function	Purpose
`LInternetResponse()`	constructor
`Response Accessors`	
`ResetResponse()`	clears out the internal storage of the object
`SetResponse()`	parses ASCII response into the internal format, this is a virtual member function
`GetResponse()`	returns an LDynamicBuffer with the text portion of the response
`GetResponseCode()`	returns the numeric code portion of the response

LInternetResponse is an abstract class. The Internet classes will automatically create the proper response object for the protocol you use. There are subclasses of LInternetResponse for each protocol implemented in PowerPlant.

The specific LInternetResponse subclasses are:

- LSMTPResponse
- LPOP3Response
- LHTTPResponse
- LFTPResponse

NOTE: The Internet classes rely on the standard C++ exception mechanism to report errors and unexpected events during operation. When a problem occurs, the classes will usually throw an LInternetResponse derived object. You should be familiar with how to use the C++ exception mechanism when using the Internet class-

es. Remember that some protocols may signal a problem that has to do with conditions of the transaction, and that you will want to check the response codes to determine if your code can work around the situation (such as remote computer being busy).

LSMTPResponse

SMTP responses use the same format as the one described in LInternetResponse. SMTP uses numeric response codes with optional text to relay the state of the transaction.

Table 8.7 Important LSMTPResponse functions

Function	Purpose
`LSMTPResponse()`	constructor
`Response Accessors`	
`SetResponse()`	parses ASCII response into SMTP response components

LPOP3Response

POP3 responses do not use a numeric status code, but rather a simple success and failure indicator. Some POP3 commands are requests for additional information. The protocol embeds the resulting information in the response following the optional text and the CRLF delimiter.

Table 8.8 Important LPOP3Response functions

Function	Purpose
`LPOP3Response()`	constructor
`Response Accessors`	
`SetResponse()`	parses ASCII response into POP3 response components

Function	Purpose
`GetStatus()`	returns the success code cast to a Boolean type
`GetResponseData()`	returns an LDynamicBuffer containing the response information (if any)

LHTTPResponse

HTTP uses numeric codes and optional text to indicate success or failure. HTTP responses are used differently than mail (SMTP and POP3) responses. In most cases, HTTP transactions are composed of a single command and response cycle. The resulting response contains the requested data (if any) or an error description which is frequently in Hierarchical Text Markup Language (HTML). LHTTPResponse builds an LHTTPMessage object to hold this data.

Table 8.9 Important LHTTPResponse functions

Function	Purpose
`LHTTPResponse()`	constructor
`Response Accessors`	
`SetResponse()`	parses ASCII response into HTTP response components
`GetStatus()`	returns true if ASCII response contains an apparently valid response code
`GetReturnMessage()`	returns an LHTTPMessage containing the response information (if any)

WARNING! Notice that the meaning of `GetStatus()` in LHTTPResponse and LPOP3Response is different. In the former, it is an indication that `GetResponseCode()` will return a supposedly valid

code number. In the latter, it returns true if the response code indicates success.

LFTPResponse

FTP responses use a numeric code and optional text to indicate the success or failure of a command. Some FTP commands are requests for additional information. The protocol embeds the resulting information in the response within the optional text area.

Table 8.10 Important LFTPResponse functions

Function	Purpose
`LFTPResponse()`	constructor
`Response Accessors`	
`SetResponse()`	parses ASCII response into FTP response components
`CommandOK()`	returns Boolean true value if response code equals FTP success code
`GetResponseData()`	returns an LDynamicBuffer containing the response information (if any)

LInternetMessage

Some Internet protocols encapsulate the data they transport inside of specially formatted messages (RFC822). A message is composed of two parts, a set of descriptive headers and a message body. Headers are composed of one or more field/value pairs. Individual protocols designate the desired order of fields in a header as well as which fields are expected and valid.

LInternetMessage is the base class that implements a storage object that models the RFC822 format. It has member functions that maintain a generic set of headers and a message body. When your program is ready to use the contents of the object, it builds a correctly formatted, complete message on demand.

Table 8.11 Important LInternetMessage functions

Function	Purpose
`LInternetMessage()`	constructor, can start from scratch or take a buffer of a previously received message
`Message Accessors`	
`ResetMembers()`	clears out message components
`SetMessage()`	parses the supplied buffer into header and body components
`GetMessage()`	builds and returns an LDynamicBuffer to a complete message
`SetHeader()`	parses the supplied buffer and store header fields
`GetHeader()`	returns LDynamicBuffer of the combined header fields
`SetMessageBody()`	stores a copy of the supplied body buffer
`GetMessageBody()`	returns LDynamicBuffer of the body
`SetArbitraryField()`	adds or changes a header field
`GetArbitraryField()`	gets a named header field if it exists

There are two subclasses of LInternetMessage:

- LMailMessage
- LHTTPMessage

LMailMessage

LMailMessage adds support for electronic mail oriented header fields. LMailMessage also provides basic MIME 1.0 functionality to support standard and multipart message bodies.

Most of the member functions manipulate specific mail header fields:

Table 8.12 Some Important LMailMessage functions

Function	Purpose
`LMailMessage()`	constructor
`Message Accessors`	
`SetTo()`	stores list of To: addresses
`AddTo()`	appends to existing To: addresses
`GetTo()`	returns LList of To: addresses
`SetCC()`	stores list of CC: addresses
`AddCC`	appends to existing CC: addresses
`GetCC()`	returns LList of CC: addresses
`SetDateTime()`	stores the message's date and time in internal format
`GetDateTime()`	returns a pointer to the internal DateTimeRec buffer
`SetFrom()`	stores a copy of the From: string
`GetFrom()`	returns a pointer to the From: string
`SetSubject()`	stores a copy of the Subject: string
`GetSubject()`	returns a pointer to the Subject: string
`SetMessageBody()`	stores a copy of the supplied body buffer, handles multipart/mixed contents
`AddMessageBodySegment()`	appends an LMailMessage to internal list of body segments
`GetMessageBodyList()`	returns an LMailMessageList containing all body parts

Function	Purpose
`MIME Management`	
`SetIsMIME()`	set true if the message uses MIME
`GetIsMIME()`	returns true if message includes MIME content
`SetBoundary()`	sets multipart boundary string
`GetBoundary()`	returns a pointer to boundary string
`SetContentType()`	sets MIME Content-Type field value
`GetContentType()`	return a pointer to content type string

LMailMessage has accessor functions for other mail related header fields. Please see the PowerPlant source code for a complete list.

NOTE: LMailMessage will support the MIME type multipart/mixed if you supply multiple message bodies, otherwise it defaults to text/plain. If you want to use another MIME type, you will need to set the MIME content type yourself and handle the type's behaviors.

LHTTPMessage

LHTTPMessage implements support for HTTP 1.0 header fields. This class implements the partial support for MIME that is expected in standard HTTP exchanges. You will need to remember to supply the correct MIME content type when you construct an LHTTPMessage object from scratch. Set the user agent field if you want to notify the remote computer of what kind of client it is communicating with. LHTTPMessage implements the basic HTTP authentication mechanism for you.

LHTTPMessage provides many functions to manipulate HTTP headers:

Table 8.13 Some Important LHTTPMessage functions

Function	Purpose
`LHTTPMessage()`	constructor
`Message Accessors`	
`SetServer()`	stores Server type string
`GetServer()`	returns a pointer to server type string
`SetUserAgent()`	stores User-Agent id string
`GetUserAgent()`	returns a pointer to User-Agent string
`SetModSince()`	sets If-Modified-Since conditional date buffer
`GetModSince()`	returns a pointer to the DateTimeRec buffer storing the If-Modified-Since date
`SetLastMod()`	sets Last-Modified resource age buffer
`GetLastMod()`	returns a pointer to the DateTimeRec buffer storing the Last-Modified date
`SetAllow()`	sets Allow string (the HTTP methods allowed)
`GetAllow()`	returns a pointer to the allow string
`SetWWWAuth()`	sets WWW-Authentication basic realm
`GetWWWAuth()`	returns a pointer to the basic realm string
`SetUserName()`	sets user name string used for authentication

Function	Purpose
`GetUserName()`	returns a pointer to the user name string
`SetPassword()`	sets password string used for authentication
`GetPassword()`	returns a pointer to the password string
`MIME Management`	
`SetContentType()`	sets MIME Content-Type field value
`GetContentType()`	returns a pointer to the content type string

LHTTPMessage has accessor functions for other HTTP related header fields. Please see the PowerPlant source code for a complete list.

TIP: HTTP is not really MIME compliant, but it uses some MIME fields to indicate how to handle message contents. You should be sure to set the content type value of LHTTPMessages you create. The default type of text/plain will probably not be the type you are usually sending.

Internet Class Utilities

You will encounter several utility classes while using any of the PowerPlant Internet classes. Two classes are particularly prevalent:

- LDynamicBuffer
- LHeaderField

LDynamicBuffer

LDynamicBuffer encapsulates a generic resizable buffer that grows or shrinks as its content changes. Since it is often difficult to predict how much data will be returned when using a number of the differ-

ent Internet protocols, this class provides a simple, efficient solution. Many of the functions that return arbitrary data in the Internet classes will return LDynamicBuffer objects.

Table 8.14 Some Important LDynamicBuffer functions

Function	Purpose
`LDynamicBuffer()`	constructor
`SetBuffer()`	stores data in the buffer, replacing previous contents
`CatBuffer()`	appends to the current buffer
`ResetBuffer()`	clears out the buffer
`GetBufferSize()`	returns the buffer size in bytes
`GetBufferH()`	returns a Mac OS Handle to the internal data buffer

LHeaderField

LHeaderField is a utility class that embodies the message header field concept described in "LInternetMessage" on page 280. This class maintains the title of the field and its value, and builds a standard header string on demand.

Table 8.15 Some Important LHeaderField functions

Function	Purpose
`LHeaderField()`	constructor
`SetField()`	parses and store the components of a complete header
`GetField()`	returns an LDynamicBuffer to a header built from the LHeaderField's title and body value
`SetTitle()`	stores the title string

Function	Purpose
`GetTitle()`	returns a pointer to the title string
`SetBody()`	stores the body's value
`GetBody()`	returns a pointer to the body value string

Other Classes

LMailMessageList, LHeaderFieldList, and UInternet are utility classes that are generally used internally to the PowerPlant Internet class library. You may need to infrequently use their capabilities. Please see the PowerPlant source code to learn details about these classes.

Implementing an Internet Enabled Application

Adding support for an Internet protocol to your application requires a simple set of tasks when you use the PowerPlant library. The Internet classes abstract away many of the mundane details of Internet programming, allowing you to focus on the actual functionality you want to implement. Much as PowerPlant relieves you of having to worry about standard user-interface issues when you use the LPane hierarchy, the Internet classes implement the underlying network code and give you an easy interface to the Internet.

When it comes to working with Internet technologies, a vast number of options are open to you. This chapter focuses on the protocols currently implemented in PowerPlant. It shows you how to use of some of the features of the Internet classes to implement simple, frequently sought features. You can use these examples as a guide in your own work, and can pattern more complicated programming on the simple examples shown here.

NOTE: The “Internet Example” sample code and this chapter’s exercise (MIST) on your CodeWarrior CD both include an imple-

mentation of a client that makes use of the protocols described in this chapter.

Many Internet protocols share the feature of one computer establishing a connection to another, and then issuing a number of commands. For each command, the receiving computer will usually send back a response. The communication between computers is often characterized as a client and server relationship (or sometimes called a consumer and producer relationship). The originating computer is the client, and its purpose is to either retrieve or send information to another program. The remote computer is often labeled the server machine. In actuality, a client machine can subsequently play the role of server, and vice-versa. Note also that it is possible for the two programs to be on the same machine, in which case, the relationship between the programs still remains the same.

Programs that send mail messages via SMTP or retrieve messages with POP3 are usually called mail clients (or user agents in mail protocol lingo). A program that gets Web pages via HTTP is a web client (or often a browser, depending on how it uses the information). An FTP client will connect to an FTP server to retrieve files to the client's computer or send files to the remote system. The PowerPlant Internet classes do not restrict you in the types of programs you can write. You can create clients, servers, or even both within the same program.

This section describes the steps you will take to create a typical client that uses an Internet protocol. The general procedure is the same regardless of the protocol you use. The following topics are included in this section:

- Choosing a Protocol
- Creating a Protocol Client
- Preparing Content
- Addressing the Remote Computer (server)
- Creating the Protocol Thread
- Creating a Connection
- Sending Content to a Server

- Receiving Responses From a Server
- Listening For Progress Messages
- Closing Down a Connection
- Handling Abnormal Conditions

Choosing a Protocol

Even before you can decide whether you are creating a client or server program, you need to understand which protocol is necessary to accomplish your desired task. If you know that you are communicating with a particular kind of server, then the choice may be trivial. If you are creating your own set of programs for both ends of a communication link, then you may need to better understand the features of the individual protocols to better match a protocol to your project. You may find that none of the protocols match your needs, and subsequently, you may need to create your own. You should note, however, that the protocols that currently exist in PowerPlant provide a rich set from which to start.

If you are dealing with electronic mail, then you will want to focus on the SMTP and POP protocol classes. You should examine the protocols to understand to what level of detail you need access. If you are sending simple messages, or want to have a simple way to check for waiting mail messages, you may be able to use the basic wrapper methods to get functionality up and running quickly. If you are creating a more complicated program--perhaps one that will act as a full featured mail agent--you will want to look at the advanced methods.

If you need to communicate with a web server, you will want to examine the HTTP classes. These classes give you access to the most common communication techniques for interacting with the servers in place today. HTTP is a simple protocol, and much of its flexibility derives from the fact that you can send pretty much any kind of message embedded within its data stream. If you need to retrieve resources from a web server, want to send feedback using an HTML Forms-like mechanism, or want to tap into other Common Gateway Interface (CGI) processing, this class will help you get started.

If you want to manipulate remote files at a more detailed level, then FTP may be the protocol you are looking for. You can retrieve remote files, send local files to the remote site, list, delete, and rename remote files, and do many other tasks required to manage a file system.

Creating a Protocol Client

Once you have decided which protocol you are going to use, you will need to build the surrounding infrastructure in your program to support the protocol classes. Generally you will want some kind of user interface that provides a way to manipulate the data that you will be interacting with on the server.

The client will be the originator of the communications link that you are creating, regardless of the protocol. The client will also be the code responsible for managing that connection. It constructs the local content that may need to be sent, properly formats the messages for transmission, gets the proper addressing information to contact the remote system, creates the communications thread, handles any errors that may occur, sends commands and data, receives responses and requested data, and cleanly closes the communications link when finished.

Preparing Content

You may choose to have your main client code create or gather the data necessary for transmission over the protocol you have chosen. The data may be text of an email message and its accompanying email addresses or it may be something as simple as responses to a dialog that you will send to an HTTP server for processing as an HTML form.

You can decide to create a fully formed message within your client code, or just collect the raw information. Minimally, you will need to know the addressing information of the remote system in order to establish the connection. Other data depends on your use of the protocol.

If you are using a protocol that will send some kind of data to the remote system, it is largely a matter of programming style and knowledge of your program's resource requirements as to whether your client just holds the raw data or creates an LInternetMessage (or derivative) based object to hold the data.

Addressing the Remote Computer

The protocols represented by the PowerPlant Internet classes are all well established Internet protocols, and as such, have assigned TCP/IP port numbers. The classes default to these standard port numbers. You will usually just need to provide an Internet address in DNS format when establishing your connection. (DNS names being in the form of “www.metrowerks.com”.) The Internet classes require that the address be in the form of a pascal string (Str255).

If your situation requires that you connect to the remote host using a port number other than the standard number, you will also have to supply that information when you create the connection.

Creating the Protocol Thread

Internet class connections rely on the threaded network mechanism of the underlying network class hierarchy. You will need to create a thread to drive each connection that you open. Depending on the complexity of the communications task, creating a thread subclass is a matter of subclassing LThread, and creating a `Run()` function that implements your task. Listing 8.1 illustrates a simple function used to request a resource via HTTP.

Once the thread is instantiated, it will handle all communications duties. You can pass a client object pointer to the thread for easier access to data stored within the client. The client object is frequently where raw data is stored until an LInternetMessage based object needs to be constructed.

Threads are usually responsible for deleting themselves once their task is complete and this model is followed when using the Internet classes. You can fire and forget the thread from your client code. If you require more interaction with the thread during a session, you

will want to establish some kind of signaling mechanism between the client and the thread. See the chapter on using the PowerPlant Thread classes for more information on how to program with threads.

If you are using a protocol to send data, such as an email message, you will want to build a properly formatted message before you start the connection. Depending on whether your client already created the message, or has been holding raw data collected from the program's user interface, the start of the thread is a good time to construct the LInternetMessage based object.

Listing 8.1 Simple HTTP Thread Run() Method

```
void *
CHTTPSendMsgThread::Run()
{
   try {
      LHTTPResponse theResponse;

   // connection will report progress to the thread
   // (this thread is an LListener)
      LHTTPConnection connection(*this);
      connection.AddListener(this);

   // constructs the message from raw data stored elsewhere
      LHTTPMessage theMessage;
      BuildMessage(theMessage);

      connection.Post("\pecho.metrowerks.com",
                      "\p/cgi/register.cgi",
                      theMessage,
                      theResponse);
      connection.RemoveListener(this);
   // connection is cleaned up as we leave try scope
  }
   catch (...) {
   // errors and problems from the Post wrapper should be
   // handled here if possible
   }
```

```
return nil;
}
```

Creating a Connection

The PowerPlant Internet classes provide a number of different ways to interact with the protocols they model. Frequently you will want to accomplish a very simple task, such as sending a single mail message, or retrieving one picture from a web server. Other times you will need to manage each command as it is sent to the remote computer, and work with each response given. To facilitate this kind of need, the classes provide both simple, "one-shot" methods, that wrap up all of the steps required to complete a task in that protocol into one function call. In some of the classes—POP and FTP in particular—you can work at a finer level of detail, and drive the session by sending individual or small sets of commands.

When you are working with the complete transaction wrappers, such as `LSMTPConnection::SendOneMessage()` or `LHTTPConnection::Get()`, a connection will be opened for you and closed after all of the task's intermediate steps have completed. When you use these commands, you do not need to worry about the state of the connection, but rather you can think on the higher level of your task.

On the other hand, all of the protocols provide the means for you to open a connection to a remote computer using the classes' `Connect()` functions. When you use this mechanism, you have a finer level of control over whether the connection stays open after a transaction completes—you can keep it open for subsequent usage. When you open a connection manually, you supply a computer's DNS address in the form of a Str255 (e.g. "\pwww.metrowerks.com") and can supply an optional port number. If you omit a port number, the default port number for the protocol is used.

Regardless of whether you then use the "one-shot" wrapper methods or the more detailed functions, the link will stay open for your use. This may be desirable for performance or other reasons.

NOTE: If you want to use some of the more advanced connection methods within a protocol—such as the individual commands within POP3 or FTP—you will have to open the connection for yourself.

Sending Content to a Server

Whether you manually opened a connection or are relying on the automatic mechanism, sending content to the remote computer is a simple matter of properly formatting the data within the appropriate LInternetMessage derived object (if applicable), and passing it to one of the messaging functions.

Depending on the protocol, you will need to supply different pieces of data when you build the message. The LInternetMessage class accumulates the data you provide as you supply it within different header fields and the message body itself. The class knows how to assemble this data into a properly formatted data buffer for transmission via the selected protocol. You can concentrate on supplying the basic information and the class will do the rest.

Receiving Responses From a Server

The protocols implemented with the PowerPlant Internet classes follow the command and response model of communication. Responses are issued by the remote computer after receiving—and possibly acting upon—a command or piece of data from the local computer. Of course, it is possible for this arrangement to be reversed, think of the response as the acknowledgment given by the receiving computer, whichever it may be.

Responses are generally of the form of some code which represents the success or failure of the action, followed by additional data, possibly including some form of embedded message. Responses in this class library are derived from the base class LInternetResponse.

The protocol wrappers shield you from much of the details of the protocol's transactions, but it is a good idea to understand the general meaning of the responses that your client can receive. You

should look at the appropriate RFCs for a listing of possible responses. In most cases, the connection classes will translate the individual protocol's response codes into the generic message codes used when broadcasting progress messages to your client.

In protocols such as HTTP, where the transaction is only composed of a single command, the response will include the requested resource or some other server generated message—often in HTML—that is supplied in the form of an LHTTPMessage. In POP3, the response when using the `GetOneMessage()` includes an LMailMessage object. Some protocols' responses are simple error codes that indicate the state of the server and the results of the command.

Listening For Progress Messages

Some function calls result in a series of commands being issued to the remote computer, or perhaps transfer large quantities of data. In these cases, the connection object will issue progress messages at each stage of the transaction. With long data transfers, some of the protocols will periodically notify you of the status of the transfer. Your use of these progress messages is optional, but they can help you keep your client abreast of the connection's state.

The progress mechanism broadcasts a generic message (see Table 8.16) and includes a pointer to the progress object. The progress structure (see Listing 8.2) includes counts of the number items (mail messages, files, etc.) that it is operating upon, and (where appropriate) how many bytes are involved with the transmission.

Table 8.16 Generic Protocol Progress Messages

Message Constant	Generic Meaning
`msg_OpeningConnection`	`Connect()` about to be called
`msg_Connected`	`Connect()` succeeded
`msg_Disconnected`	`Disconnect()` succeeded
`msg_SendingData`	Sending data in progress
`msg_ReceivingData`	Receiving data in progress

Message Constant	Generic Meaning
`msg_SendItemSuccess`	Successfully sent one item
`msg_SendItemFailed`	Failed sending one item
`msg_RetrieveItemSuccess`	Successfully received one item
`msg_RetrieveItemFailed`	Failed receiving one item
`msg_DeleteItemSuccess`	Successfully completed a delete command
`msg_DeleteItemFailed`	Failed completing a delete command
`msg_SendingItem`	Starting to send an item
`msg_ReceivingItem`	Starting to receive an item
`msg_ClosingConnection`	`Disconnect()` about to be called

You should note that the message transmitted by the `BroadcastProgress()` function is a standard code that is used in all protocols to indicate what the Internet class library is currently doing. The actual meaning of the codes are somewhat context sensitive, some make more sense in certain protocols than in others. The code does not necessarily reflect the protocol specific state or response code of the active protocol. For that, you should refer to any LInternetResponse derived objects you may receive either directly via the various function calls or indirectly by the exception mechanism.

Listing 8.2 The Progress Structure

```
struct SProgressMessage {
   LInternetProtocol *theProtocol;// src protocol object
   LStr255   currentItem;         // descriptive text for item
                                  //   currently being manipulated
   UInt32    totalItems;          // total items involved
                                  //   in transaction
   UInt32    completedItems;      // completed count in items
   UInt32    totalBytes;          // total bytes in transaction
```

```
   UInt32    completedBytes;      // completed count in bytes
};
```

To receive progress messages, you need to attach an LListener to the LInternetProtocol you want to monitor. You will usually do this when you create the protocol object. See Listing 8.1 for an example.

Closing Down a Connection

If you are using the simple wrapper functions for the LInternetProtocol object in your program, and if you haven't manually opened the connection, the protocol object will handle the disconnection and cleanup of your communication link for you.

If you have manually opened the connection with `Connect()`, the wrapper functions will not close down the connection. Use `Disconnect()` to accomplish this task.

You should be sure to capture any exceptions during the use of both the simple wrapper functions and the more detailed protocol functions. Most of the communication-oriented functions do not close the connection on an exception, so you will need to do that yourself.

If you just destroy the LInternetProtocol object, its LEndpoint object will be destroyed, but the proper protocol rundown procedure does not occur. The correct closing sequence for each protocol is implemented in its `Disconnect()` member function.

Handling Abnormal Conditions

When a problem occurs during the operation of a protocol, such as an error message being returned from the remote computer, the LInternetProtocol object will throw an appropriate LInternetResponse based object. You should usually get into the habit of surrounding your networking code with `try` and `catch` blocks to capture these exceptions. Many times, you may be able to correct the problem, or at least notify your user of the problems existence, and then carry on with the operation.

The Internet classes are built on top of the PowerPlant network classes and this leads to another source of possible exception codes. The network classes will throw exceptions when they encounter errors in the operating system networking code. In most cases, these exceptions will be propagated up through the Internet classes. You should be prepared to catch and handle these exceptions in your code.

Serious errors coming from programming mistakes within your client code will also frequently cause exceptions to be thrown. When a protocol detects that it is in an illegal state for the operation you are attempting, it will throw an exception based on the boolean result comparing the current state with the expected state. One example of this is when you forget to open a connection (or the connection doesn't open properly) and you proceed to try to use a function that sends data.

TIP: You may want to have separate `catch` blocks for the different kinds of exceptions thrown by the Internet classes. The first could catch LInternetResponse objects, the second could check for exceptions such as PowerPlant network class error codes or false assertions, and a third should catch all other exceptions.

Summary of Internet Protocol Usage in PowerPlant

Adding Internet awareness to your programs is becoming more and more important each day. Whether you want to be able to register your user over the Internet by taking advantage of your Web site, or you want the user to send technical support questions via electronic mail, there are an unlimited number of capabilities you can add to your application by taking advantage of just a few standard Internet protocols.

The PowerPlant Internet classes provide implementations of the most frequently used protocols on the Internet today. By giving you access to HTTP, SMTP, POP3, and FTP, you can concentrate on in-

venting new features for your program that take advantage of this foundation, and you can avoid the drudgery of constantly reinventing the wheel.

Code Exercise

The Mini Internet Support Tool program (MIST) illustrates how to build selective Internet protocol functionality into your application to add common support features. It shows one use of each of the main protocols provided by the PowerPlant Internet class library. Although there are many different ways to use each protocol, this example should give you a basic understanding of the classes, and how simple it is to add Internet capabilities.

The purpose of this exercise is to give you experience using the Internet classes in PowerPlant. You will learn about some of the more common member functions and how to format, send, and receive Internet messages. This exercise does not delve into the details of the individual Internet protocols, nor is it a tutorial on network programming in general. You should examine the topic "Where to Learn More About Internet Protocols" on page 254 for that kind of information.

In this exercise, you will create portions of the Mini Internet Support Tool program. As you gain experience with the Internet classes, you will notice that many of the steps to add a protocol into your program are similar for each of the other protocols.

Before we begin the exercise, there are some issues you should note. This code is structured in a simple manner to make it easier to understand the steps needed to add Internet functionality into your application. The code avoids heavy error checking and does not capture all of the possible return results from the remote computer. This exercise also makes use of the precompiled header feature of PowerPlant, so you will notice that header files for non-Internet classes are not included in most files.

Let's look at the steps you will take to add simple Internet functionality. In this exercise you will write code to:

- Create a thread to run HTTP. This allows the application to continue operating while the protocol does its work.
- Build an HTTP message. This is the data you will send to the web server.
- Post HTTP data to a web server. This is where you actually send the message.
- Check the response code. This is how to verify your data was processed correctly.
- Create a thread to run SMTP. You will use SMTP to send electronic mail.
- Build a mail message. This is the format you will use for SMTP and POP3.
- Send a mail message with an SMTP wrapper function. This is the easiest way to send one mail message at a time.
- Capture an exception generated by the connection. This is how you are notified of problems with an Internet protocol.
- Retrieve mail message headers via POP3. This is the way you can grab just the header part of mail messages via POP3.
- Scan mail messages for arbitrary header fields. This is the way you can look for header fields other than those directly support by the Internet classes.
- Retrieve a directory listing via FTP. This is the way you can scan a remote computer's file system for a file to download.
- Download a remote file. This is how you can retrieve an arbitrary data file using one of the FTP wrapper functions.

1. Create a thread to run HTTP.

`SendRegistrationMessage()` `CRegisterViaHTTP.cp`

The protocol code for each of the Internet classes expects to run from within the body of an LThread derived object. You must subclass a thread object and override its `Run()` method to make your calls to the remote computer. You will usually instantiate the thread object from within your client code and use `LThread::Resume()` to get it started.

As usual, the existing code is in italic.

```
mThread = nil;
mThread = new CRegisterViaHTTPThread(this);
if (mThread)
   mThread->Resume();
```

2. Build an HTTP message.

`BuildMessage()` — `CRegisterViaHTTP.cp`

In the HTTP protocol, data is exchanged between computers using LHTTPMessages to package the information into the format specified in the protocol's RFC. When you create an empty LHTTPMessage object, you need to populate the minimum header fields for the operation you are trying to complete.

MIST collects information from the user to send to an HTTP server to complete an on-line registration form. The MIST program mimics the operation of a Web browser's form capabilities by using an HTTP POST command to send data that has been specially formatted using an HTML form encoding.

After setting the body data you need to set the content type of the data, and to be helpful to the Web server, the type of client program sending the data.

```
theMessage.SetMessageBody(*theFormH,theLength);
theMessage.SetUserAgent("MIST");
theMessage.SetContentType("application/x-www-form-urlencoded");
```

The LHTTPMessage automatically sets other headers when you supply body data to correctly identify the length of the message.

3. Post HTTP data to a web server.

`Run()` — `CRegisterViaHTTP.cp`

Your thread is now running and has some data to transmit to the Web server. You need to create the protocol object, and tell it to send the data using the wrapper function `LHTTPConnection::Post()`.

```
LStr255 theHost(STRx_HTTP, str_HOST);
LStr255 theResource(STRx_HTTP, str_URL);
LHTTPResponse theResponse;
LHTTPConnection connection(*this);
connection.AddListener(this);
connection.Post(  theHost,
                  theResource,
                  theMessage,
                  theResponse);
```

4. Check the response code.

`DisplayResponse()` — `CRegisterViaHTTP.cp`

When the POST command completes, it returns a status value from the remote computer in the LHTTPResponse object. You should check the return value to determine the success or failure of the POST command. An HTTP transaction is composed of one command and response cycle. If you receive a status code that indicates an error, you will need to reissue the command after possibly correcting the message contents, host, or resource specifier.

```
Int32 theResponseCode;
theResponseCode = theResponse->GetResponseCode();
if (kHTTPRequestOK == theResponseCode ||
    kHTTPRequestNoResponse == theResponseCode)
   mProgress->SetDescriptor(
      LStr255("Registration transmission was a success"));
else
   mProgress->SetDescriptor(LStr255("Failure"));
```

5. Create a thread to run SMTP.

`SendRegistrationMessage()` — `CSendQuestionSMTP.cp`

Like the other protocol objects in the Internet class library, you need to create an LThread based object to run the SMTP transaction. Once

the client code has gathered up all of the desired data, create your thread, and start it running by calling its `Resume()` function.

```
mThread = nil;
mThread = new CSendQuestionSMTPThread(this);
if (mThread)
   mThread->Resume();
```

6. Build a mail message.

`BuildMessage()` — `CSendQuestionSMTP.cp`

Electronic mail messages are modeled by LMailMessage objects. Like the LHTTPMessage class, LMailMessage adds support for setting the most frequently used header fields geared towards email. LSMTPConnection sends LMailMessages to the destination addresses you provide. Minimally, you need to specify a To and From address to send a mail message. It is helpful to add a Subject to the message as it both helps the recipient know what the mail message is about, and the Subject string is used by LSMTPConnection's progress mechanism to notify you of which message it is sending. In this exercise, you need to add a special header field "X-Mailer" so that you can tag the message with the type of email client that was used to send the message. You can use the `LInternetMessage::SetArbitraryField()` function to add any field that doesn't have its own specific access function.

```
ThrowIfNot_(mExample->GetEmail(anEmailAddr));
p2cstr(anEmailAddr);
theMessage.AddTo((char *) anEmailAddr);
theMessage.SetFrom((char *) anEmailAddr);
theMessage.SetSubject("MIST: Tech Support Question");
theMessage.SetArbitraryField("X-Mailer", "MIST");

ThrowIfNot_(mExample->GetQuestion(aQuestionH));
Uint32 theQuestionSize = ::GetHandleSize(aQuestionH);
StHandleLocker theLock(aQuestionH);
theMessage.SetMessageBody(*aQuestionH, theQuestionSize);
```

7. Send a mail message with an SMTP wrapper function.

`Run()` `CSendQuestionSMTP.cp`

The bulk of the code to implement a protocol's transaction is frequently located in the thread's `Run()` function. Once you have constructed your LMailMessage and decided on its destination, you can make use of the `LSMTPConnection::SendOneMessage()` wrapper function. By using the simpler API of the wrapper functions, you avoid all of the complicated details of running the protocol.

```
LSMTPConnection connection(*this);
connection.AddListener(this);
connection.SendOneMessage( theHost, theMessage);
connection.RemoveListener(this);
```

8. Capture an exception generated by the connection.

`Run()` `CSendQuestionSMTP.cp`

The LInternetProtocol derived classes report status and error information using two principal methods. They will throw an exception when an unexpected event or error occurs and they can report general status information through the progress mechanism.

You should be sure to place `try` and `catch` blocks around your protocol function calls to handle the exceptions generated by them.

```
catch (ExceptionCode err) {
   SysBeep(30);
   if (err_AssertFailed == err)
      DisplayProgress("Message was not sent, no connection");
   else
      DisplayProgress("Connection failed due to unknown reason");
}
```

If you are using the progress mechanism, you will need to add the appropriate test for the progress message error codes in your thread's `ListenToMessage()` function. LSMTPConnection will re-

port a `msg_SendItemFailed` status code if the LMailMessage can't be sent.

```
switch (inStatusCode) {
   case msg_SendItemFailed:
      sprintf(statusMessage, "Failed to send: %#s",
              theMsg->currentItem);
      DisplayProgress(statusMessage);
      SysBeep(30);
      break;
   default:
      break;
  }
```

9. Retrieve mail message headers via POP3.

`Run()` `CCheckPOP.cp`

You can make use of the various LInternetProtocol derived classes' wrapper functions to help you make decisions about more sophisticated processing. Suppose that you want to selectively handle email messages depending upon the kind of mail client that was used to send the message. You can retrieve mail headers without having to download the potentially bulky message bodies by using the `LPOP3Connection::GetHeaders()` function.

```
LMailMessageList theHeaderList;
POP3Connection connection(*this);
connection.AddListener(this);
connection.GetHeaders(
   theHost,
   theUsername,
   thePassword,
   &theHeaderList,
   mExample->GetUseAPOP());
connection.RemoveListener(this);
```

The `GetHeaders()` function returns an LMailMessageList of LMailMessage objects. These LMailMessage objects do not contain the body portion of the mail message.

10. Scan mail messages for arbitrary header fields.

`ScanListForMIST()` — `CCheckPOP.cp`

Now you can scan through the list of LMailMessage headers looking for messages sent using the MIST client. A header field that describes the type of email client that sent a message is not always supplied. Furthermore, the RFCs which define the format of mail messages do not include a standardized header field that gives this kind of information. Therefore, you will look for an experimental field called "X-Mailer" which you always provide in any outgoing mail messages sent by the MIST client. You use the `LInternetMessage::GetArbitraryField()` function to find fields that are not supported directly by the Internet classes' access functions.

```
Int32 msgCount = 0;
LHeaderField tmpField;
LMailMessage *currMsg;
LArrayIterator iter(theMsgList);
while(iter.Next(&currMsg))
{
   if (currMsg->GetArbitraryField("X-Mailer", &tmpField))
   {
      if (0 == strcmp("MIST", tmpField.GetBody()))
         ++msgCount;
   }
}
```

11. Retrieve a directory listing via FTP.

`CLoadListFTPThread::Run()` — `CRetrieveUpdateFTP.cp`

Suppose you have released a series of update files for your user and you would like her to pick a file from a list that best meets her needs. You can cull such a list from the files stored on your FTP server. By using the `LFTPConnection::ListFolder()` function, you

gather the names of all of the files in a directory, and can use that data to build a pick list in your user interface.

```
progress.currentItem = theHost;
connection.BroadcastProgress(msg_OpeningConnection,
                             progress, true);
connection.Connect((ConstStr255Param) theHost);
connection.BroadcastProgress(msg_Connected,
                             progress, true);

connection.ListFolder(&theListBuffer, p2cstr(theRemoteDir), true);

progress.currentItem = theHost;
connection.BroadcastProgress(msg_ClosingConnection,
                             progress, true);
connection.Disconnect();
connection.BroadcastProgress(msg_Disconnected,
                             progress, true);
```

Because `ListFolder()` is not a wrapper function, you will need to initiate the remote connection yourself. You might want to keep the connection open after this call in anticipation of the user selecting a file from the list and wanting to start a download. If you follow the leave-it-open strategy and later use one of the wrapper functions to download a file, you will have to remember to close the connection manually. The wrapper functions leave connections in the state they find them in. In this example, we are closing the connection so that later use of a wrapper function will reopen the link.

TIP: Notice the calls to `BroadcastProgress()` in the `CLoadListFTPThread::Run()` function. Wrapper functions contain calls to the progress mechanism to notify your code of the state of a connection. However, if you manually open and close a connection, and use various utility routines in-between, you will have to insert your own calls to the `BroadcastProgress()` function if you want your code to generate progress messages.

12. Download a remote file.

`CRetrieveFTPThread::Run()` `CRetrieveUpdateFTP.cp`

Now that a file has been picked for downloading, it is really easy to initiate a file retrieval operation using the wrapper function `LFTPConnection::GetFile()`. In this example, we are using the username "anonymous" with a password that is equivalent to our email address. This is the standard access information for a guest connection on many FTP servers.

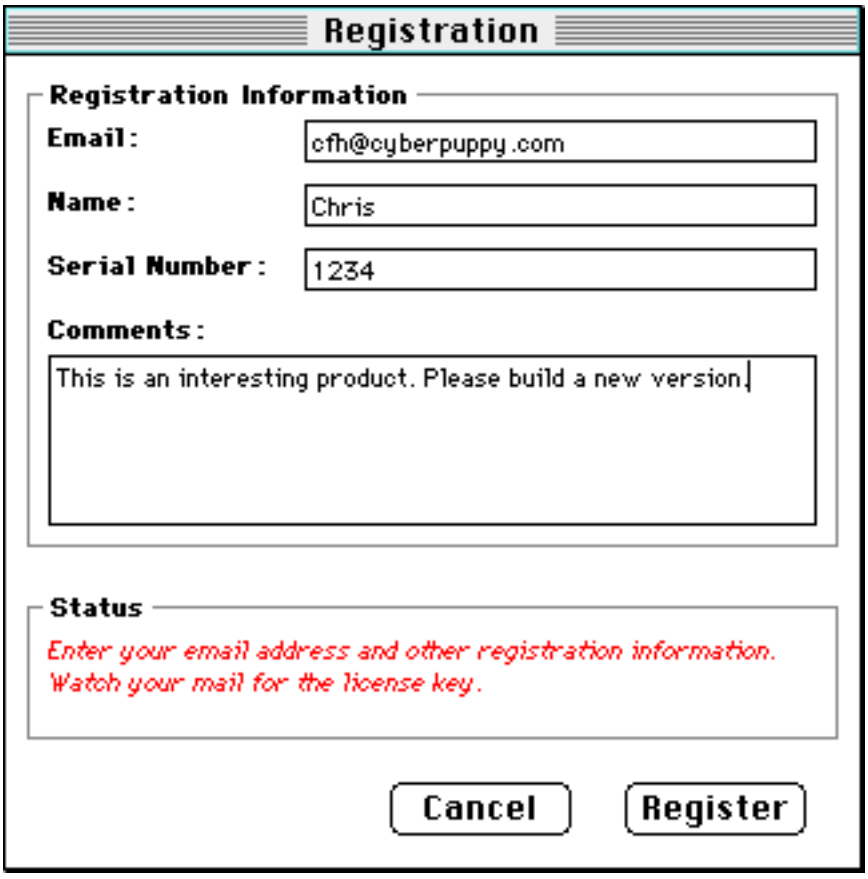
```
connection.AddListener(this);
connection.GetFile( (ConstStr255Param) theHost,
                    "anonymous", "MIST@metrowerks.com", "",
                     FTPASCIIXfer, p2cstr(theRemoteFile),
                     &theSaveFile);
connection.RemoveListener(this);
```

13. Build and run the application.

After the application compiles and runs successfully, you will be able to make one of four choices in the **Tools** menu.

The **Register…** command brings up the window shown in Figure 8.2. Enter your correct Internet mail address in the email field, and any other values for the rest of the form. When you press the Register button, the window's contents are packaged into an LHTTPMessage and transmitted to the remote computer (a Web server at Metrowerks, in this case). If you supplied a correct email address, a mail message will be sent to you containing a display of the values you supplied in the form.

Figure 8.2 Registration using HTTP



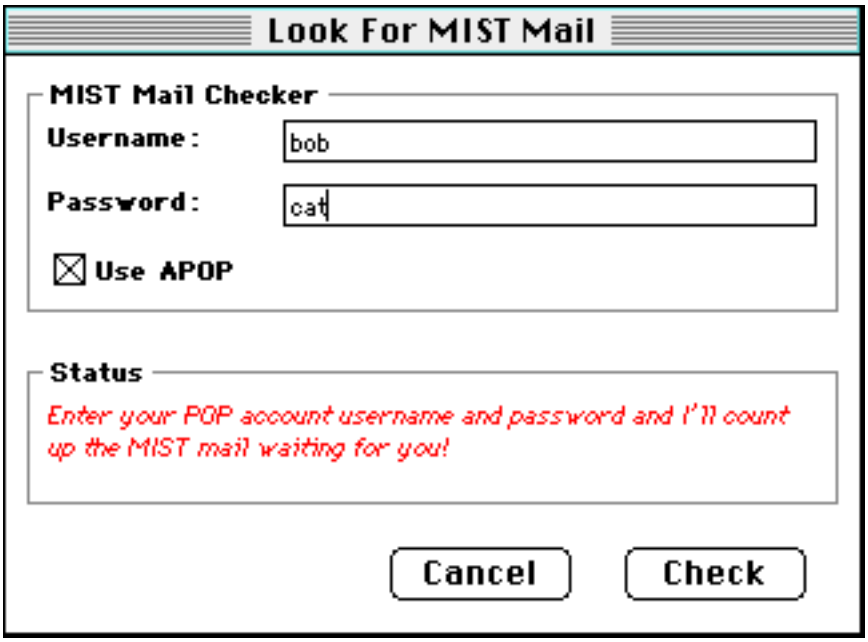
To use the next two tools, first go to the **Preferences…** command (found under the **Edit** menu). Enter your SMTP and POP3 DNS host names in the dialog and press OK. You will now be ready to run the rest of the exercise code.

You can choose the **Contact…** menu item to send a simulated technical support question via SMTP. The window in Figure 8.3 requires that you enter your own email address and a question. (If you have set the appropriate server addresses in the Preferences dialog, the message you send will be echoed back to you by the mail server.) You could use a mail feature like this in one of your programs to automatically send questions to a well know email address (such as technical support) from within your application.

Figure 8.3 Technical Support Question via SMTP

Next, if you select the **Check...** menu item, you will be presented with a window as shown in Figure 8.4. Here you should enter the POP3 user name and password for your email account. If your POP server is appropriately specified in the Preferences dialog, the program will log on to your POP server and it will scan your mailbox for any messages that were sent to you from the MIST client. You can send one or two messages from either the Contact Tech Support window or the Register window to give the scanner something to look for. The code demonstrates a technique for selectively finding and operating upon specific mail messages waiting on a remote server.

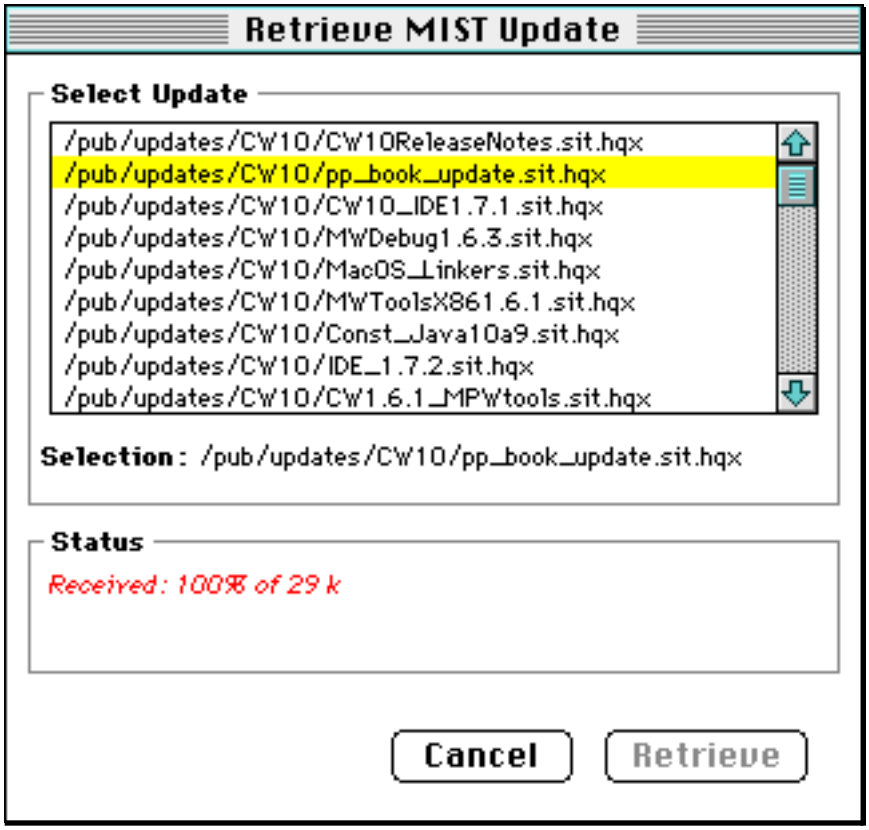
Figure 8.4 Checking for MIST Mail via POP3



Lastly, if you select the **Retrieve Update…** menu item, you will see a window as shown in Figure 8.5. If MIST can connect to the remote FTP server, it displays a list of files in the target working directory. If you double-click on a file from the list and press the **Retrieve** button, MIST will download the file to your local hard drive. The code demonstrates a technique for finding and retrieving specific files on a remote server. You could use this technique in your own program to provide a way for the user to automatically get updates or enhancements to your application.

MIST is a simple collection of examples whose main purpose is to illustrate just how easy it is to use the Internet protocols as implemented in PowerPlant. With this new arsenal of tools, you should be communicating across the Internet in no time!

Figure 8.5 Downloading a file via FTP



9

Offscreen Drawing in PowerPlant

This chapter discusses how to use PowerPlant for drawing to an offscreen graphics environment, a process also known as double buffering.

Introduction to Offscreen Drawing in PowerPlant

From the user's perspective, a screen update can be a painful experience. In a traditional approach, an application draws directly to the screen. In effect, you create the image piecemeal before the eyes of the user. As a result, updates can cause interesting and unintended visual effects. Objects may flicker needlessly, for example. In a crowded window with lots of objects, drawing may seem slow.

One solution to this problem is to create an offscreen graphics environment. Rather than draw to the screen, your application draws offscreen. When the drawing is complete, you move the finished image to the screen in one piece, a process known as blitting.

The offscreen drawing process requires slightly more time, because you must first draw the image and then blit it to the screen. However, the image appears all at once. As a result, the user perceives drawing as much faster. The user sees the end result, not the process that creates the result.

The difficulty in implementing offscreen drawing on your own is that you must do a lot of background work to create and manage the offscreen environment. PowerPlant takes care of most of the details for you, making offscreen drawing simple and straightforward.

In this chapter we discuss how you can implement offscreen drawing in a PowerPlant application. The topics discussed include:

- Offscreen Drawing Strategy—the PowerPlant approach to offscreen drawing
- Offscreen Drawing Classes—the individual PowerPlant classes involved in offscreen drawing
- Implementing Offscreen Drawing in PowerPlant—working with PowerPlant's offscreen drawing classes

Offscreen Drawing Strategy

As you know, an offscreen graphics environment in the Mac OS is called a GWorld. PowerPlant's classes related to offscreen drawing are very simple. They hide the entire Toolbox API related to GWorlds. As a result, implementing offscreen drawing is extremely simple.

When dealing with the Toolbox directly, GWorlds have many permutations. You can specify that they be created in a specified pixel depth; in the application heap or temporary memory; with a custom color table or the default color table; with a custom GDevice or not; as a purgeable item or not; and so on. PowerPlant lets you manipulate all these features. You may specify parameters to constructors to set up the GWorld to your liking.

However, the simple truth is that most of the time, you just want very straightforward behavior. You want a pixel depth equal to the deepest monitor involved in drawing. You use a default color table, default GDevice, and so on. In PowerPlant, the necessary constructors all have default arguments that set up a default GWorld simply and effectively. As a result, unless you're doing something tricky or unusual, you don't have to worry about these details. PowerPlant takes care of them for you.

PowerPlant gives you two more choices. You can easily create a typical GWorld that persists as long as you want. You can also create a temporary GWorld that lasts for just one function. Although this might seem like an odd thing to do at first glance, a temporary GWorld can be very useful.

Offscreen Drawing Classes

There are three classes in PowerPlant related to offscreen drawing. They are:

- LGWorld—implements a classic GWorld
- StOffscreenGWorld—implements a temporary GWorld
- LOffscreenView—draws its contents to a temporary offscreen GWorld

These classes are not related to each other hierarchically. Both LGWorld and StOffscreenGWorld are completely independent classes that can be used without any other part of PowerPlant. LOffscreenView is a member of the PowerPlant pane classes, and as such is dependent upon the core of PowerPlant being present.

The source code for these classes is fully commented, and you should read those comments for further insight.

LGWorld

LGWorld is a simple PowerPlant class. It is declared in `UGWorld.h` and defined in `UGWorld.cp`. The purpose of LGWorld is to create and manage a GWorld.

This class has four data members, shown in Table 9.1.

Table 9.1 LGWorld data members

Data member	Stores
`mMacGWorld`	pointer to the GWorld
`mBounds`	bounds of the GWorld
`mSavePort`	save the current graphics port
`mSaveDevice`	save the current GDevice

In general you won't have to deal with these data members directly. The `mMacGWorld` data member is set by the constructor when the

GWorld is created. You can retrieve this value with an accessor function, `GetMacGWorld()`. You set the initial bounds of the GWorld in the class constructor. The class functions save and restore the graphics port and GDevice when necessary.

Table 9.2 lists every member function in LGWorld.

Table 9.2 LGWorld member functions

Function	Purpose
`LGWorld()`	create GWorld
`~LGWorld()`	release GWorld memory
`BeginDrawing()`	begin drawing to GWorld
`EndDrawing()`	end drawing to GWorld
`CopyImage()`	copy image to screen
`GetMacGWorld()`	return pointer to GWorld
`SetBounds()`	set bounds of the GWorld

The constructor is perhaps the most interesting function. It does all the work of setting up the GWorld. Here's the prototype from the header file.

```
LGWorld(const Rect &inBounds,
        Int16 inPixelDepth = 0,
        GWorldFlags inFlags = 0,
        CTabHandle inCTableH = nil,
        GDHandle inGDeviceH = nil);
```

The only parameter you *must* provide is the bounds (in pixels, in local coordinates) of the GWorld you desire. You may pass in other values as you wish, but most of the time you can rely on the default values declared in the prototype. A zero pixel depth means that the GWorld uses the maximum depth of all screen devices intersected by the bounds. A `GWorldFlags` value of zero means that the GWorld will be created in the application heap (among other

things). Nil values for the color table and the GDevice mean you use the default color table and GDevice. For information on how you can manipulate these parameters to create various kinds of GWorlds, you should refer to *Inside Macintosh: Imaging with QuickDraw*, page 6-16.

Because of the design of the LGWorld constructor, most of the time all you have to do is tell PowerPlant how big you want the GWorld to be. PowerPlant takes care of the rest.

The remaining functions are very simple. The destructor releases GWorld memory. `BeginDrawing()` saves the current port and GDevice, sets the port for the GWorld, and locks the GWorld pixels in place. `EndDrawing()` unlocks the pixels and restores the saved port and GDevice. In between calls to `BeginDrawing()` and `EndDrawing()`, you do your drawing. Any drawing appears in the GWorld, not on screen.

WARNING! Every call to `BeginDrawing()` must be balanced with a corresponding call to `EndDrawing()`.

The `CopyImage()` function uses the Toolbox `CopyBits()` function to blit the GWorld to the screen. Once again, it is worth looking at the prototype.

```
void CopyImage(GrafPtr inDestPort,
               const Rect &inDestRect,
               Int16 inXferMode = srcCopy,
               RgnHandle inMaskRegion = nil);
```

You provide the destination port to which the image is going, and the destination rectangle. The destination rectangle is typically the same size as the GWorld, but can vary if you want to scale the image. The transfer mode and mask region have default parameters, so you do not need to provide them if the default values suit your purpose.

LGWorld Limitations

The LGWorld functions do not contain a single call to the Toolbox routine `UpdateGWorld()`. This can be a significant problem for a quality implementation of offscreen drawing.

You should call `UpdateGWorld()` after every update event, whenever the GWorld changes size, and whenever the destination window moves or changes size. These events can cause pixel depth to change if the window crosses multiple monitors.

For example, you may modify the bounds of a GWorld after you create it. Use the accessor function `LGWorld::SetBounds()`. However, `SetBounds()` doesn't work right because it doesn't call `UpdateGWorld()`.

Listing 9.1 shows a better way to set a GWorld's bounds.

Listing 9.1 A better SetBounds() function

```
void MyGWorld::SetBounds(const Rect& inBounds)
{
  mBounds = inBounds;

  GWorldFlags flags = ::UpdateGWorld(&mMacGWorld,
                        0, inBounds, NIL, NIL, 0);

// From original LGWorld
  ::GetGWorld(&mSavePort, &mSaveDevice);
  ::SetGWorld(mMacGWorld, nil);
  ::SetOrigin(mBounds.left, mBounds.top);
  ::SetGWorld(mSavePort, mSaveDevice);
}
```

You would of course pass your desired pixel depth to the `UpdateGWorld()` function. You should also check for errors.

There is yet another significant limitation to LGWorld. If you look at the class declaration in `UGWorld.h`, you will see that none of the

functions in LGWorld are virtual. Therefore, you cannot use LGWorld as a base class and derive your own classes from it.

If you wish to extend the functionality of LGWorld, you should simply copy the code from LGWorld into your own class. Then modify your class as you wish, and substitute your class for LGWorld.

Even with these limitations in mind, (incorrect GWorld updating and no way of subclassing LGWorld), LGWorld is a simple wrapper for the Mac OS offscreen graphics worlds. All the complexity is reduced to four functions:

- a constructor to create the GWorld, requiring one parameter
- a call to prepare the GWorld for drawing
- a call to close the GWorld for drawing
- a call to move the image to the screen

You can use LGWorld in any C++ code. It is completely independent of the rest of PowerPlant.

StOffscreenGWorld

StOffscreenGWorld is a simple PowerPlant class. It is declared in `UGWorld.h` and defined in `UGWorld.cp`. It is similar to, but simpler than LGWorld. Like LGWorld, it creates and destroys a GWorld. Unlike LGWorld, the GWorld created by StOffscreenGWorld exists for the duration of one function.

StOffscreenGWorld is a stack-based PowerPlant class. The constructor creates a GWorld when you instantiate an StOffscreenGWorld variable. The GWorld continues to exist only as long as the local variable holding the StOffscreenGWorld object remains in scope. When the variable goes out of scope, the StOffscreenGWorld destructor copies the GWorld image to the screen, and destroys the GWorld.

StOffscreenGWorld has the same data members as LGWorld, and they serve the same purposes.

Table 9.3 StOffscreenGWorld data members

Data member	Stores
`mMacGWorld`	pointer to the GWorld
`mBounds`	bounds of the GWorld
`mSavePort`	save the current graphics port
`mSaveDevice`	save the current GDevice

In general you won't have to deal with these data members directly. The `mMacGWorld` data member is set by the constructor when the GWorld is created. You can retrieve this value with an accessor function, `GetMacGWorld()`. You set the initial bounds of the GWorld in the class constructor. Unlike LGWorld, you cannot modify the bounds afterwards. The class functions save and restore the graphics port and GDevice whenever necessary.

There are only three member functions in StOffscreenGWorld.

Table 9.4 StOffscreenGWorld member functions

Function	Purpose
`StOffscreenGWorld()`	create GWorld and set up for drawing
`~StOffscreenGWorld()`	copy image to screen and release GWorld memory
`GetMacGWorld()`	return pointer to GWorld

The constructor and destructor do all the work. Here's the prototype for the constructor from the header file. The parameters are identical to those in LGWorld, and they serve the same purposes.

```
StOffscreenGWorld(const Rect &inBounds,
                  Int16 inPixelDepth = 0,
                  GWorldFlags inFlags = 0,
```

```
CTabHandle inCTableH = nil,
GDHandle inGDeviceH = nil);
```

The only parameter you must provide is the bounds (in pixels, in local coordinates) of the GWorld you desire. You may pass in other values as you wish, but most of the time you can rely on the default values declared in the prototype. A zero pixel depth means that the GWorld uses the maximum depth of all screen devices intersected by the bounds. A `GWorldFlags` value of zero means that the GWorld will be created in the application heap (among other things). Nil values for the color table and the GDevice mean you use the default color table and GDevice. For information on how you can manipulate these parameters to create various kinds of GWorlds, you should refer to *Inside Macintosh: Imaging with QuickDraw*, page 6-16.

The destructor restores the current port to what it was when the StOffscreenGWorld was created, copies the offscreen image to that port, and then destroys the GWorld.

WARNING! When you call the StOffscreenGWorld constructor, the current port must be the destination of the offscreen image. The StOffscreenGWorld destructor copies the image to the port that was current when the StOffscreenGWorld object was created.

You can use StOffscreenGWorld in any C++ code. It is completely independent of the rest of PowerPlant. LOffscreenView uses StOffscreenGWorld.

LOffscreenView

The LOffscreenView class is a descendant of LView in the LPane class hierarchy. In fact, this class does not do any offscreen drawing directly. It uses StOffscreenGWorld to handle drawing.

In typical use, you don't have to modify or call any function or feature of LOffscreenView. Here's how it works.

Assume you have a complex set of objects you wish to draw to an offscreen graphics environment. You want to do this so that updates appear instantaneous and the user doesn't see each individual object drawn independently.

In PowerPlant's visual interface builder, Constructor, you can add an LOffscreenView to the window. Then put all the objects you wish drawn offscreen into that view.

When PowerPlant attempts to draw the view, it calls the LOffscreenView's `Draw()` function. `LOffscreenView::Draw()` sets up an StOffscreenGWorld variable. From that point on, all the contents of the view draw into the GWorld, not the screen.

After all the contents of the view have drawn themselves, control returns to `LOffscreenView::Draw()`. The StOffscreenGWorld variable goes out of scope, and the offscreen image is blitted to the screen.

Examine the source code for `LOffscreenView::Draw()` to see how this works. The function first attempts to build the GWorld in temporary memory. It is a short-lived GWorld, so this is an appropriate use of temporary memory. If that fails, it allocates the GWorld out of the application's own memory space. If that fails, the objects draw directly to screen.

You get all the benefits of offscreen drawing automatically! Simply place your panes inside an LOffscreenView in Constructor. You can do this for the entire contents of a window, or for one or more parts of a window. The choice is yours. The rest is automatic and free.

The only negative side is that there is a minor time penalty because the GWorld is created and destroyed every time `LOffscreenView::Draw()` is called. If this is a significant issue to you, consider designing your application to use LGWorld instead.

Implementing Offscreen Drawing in PowerPlant

There are, essentially, three strategies you may follow, depending upon your individual needs.

- Using LOffscreenView—and use StOffscreenGWorld
- Using StOffscreenGWorld Directly
- Using LGWorld

Using LOffscreenView

To use LOffscreenView, use Constructor to place your panes inside the LOffscreenView. After that, your work is done. You'll do this in the code exercise for this chapter.

Using StOffscreenGWorld Directly

To use StOffscreenGWorld directly, simply instantiate an StOffscreenGWorld object at the appropriate moment, before the drawing code. When the StOffscreenGWorld variable goes out of scope, the drawing will be blitted to the screen. Listing 9.2 shows a sample of how easy it is.

Listing 9.2 An StOffscreenGWorld sample

```
void MyPane::DrawSelf()
{
  Rect frame;
  CalcLocalFrameRect(frame);

  // Allocate offscreen GWorld where subsequent
  // drawing operations will take place
  StOffscreenGWorld offWorld(frame);

  // draw pane
}
```

By the simple expedient of creating an StOffscreenGWorld variable, all pane drawing occurs in the offscreen world. When the pane's `DrawSelf()` function goes out of scope, the pane's image is blitted to the screen by the StOffscreenGWorld destructor.

WARNING! There is a danger here. In a program that uses C++ exceptions, if an exception occurs during the drawing process, the StOffscreenGWorld destructor is called anyway to blit the image to screen. However, the exception might be of such a nature that you do not want to draw the image. If this case applies to you, you should not use StOffscreenGWorld. You should use LGworld, or create your own class that handles exception situations.

Using LGWorld

Using LGWorld is a little more complicated, but not much. A typical use would be to create a new LGWorld object (in the application's heap) in a pane's constructor. You store the pointer to the LGWorld object in a data member. You delete the LGWorld object in the pane's destructor.

You use the LGWorld to store the visual image of the pane. You must explicitly call `BeginDrawing()` before drawing offscreen, and `EndDrawing()` when done. Use `CopyImage()` to move the offscreen image to the screen. In Listing 9.3, a sample pane class uses LGWorld to maintain its contents on a long-term basis.

Listing 9.3 A typical use of LGWorld

```
MyPane::MyPane() { // constructor creates GWorld
  Rect frame;
  CalcLocalFrameRect(frame);
  mGWorld = new LGWorld(frame, 8); // 8-bit depth
}

MyPane::~MyPane() { // destructor deletes GWorld
  delete mGWorld;
}

MyPane::DrawSelf() { // copy GWorld image to screen
  Rect frame;
  CalcLocalFrameRect(frame);
  mGWorld->CopyImage(GetMacPort(), frame);
}
```

```
MyPane::AddRectToImage(Rect &inRect) {
  mGWorld->BeginDrawing(); // draw offscreen
  PaintRect(&inRect); // add new rectangle
  mGWorld->EndDrawing(); // end offscreen drawing

  Rect pRect = inRect; // update new drawing
  LocalToPortPoint(&topleft(pRect));
  LocalToPortPoint(&botRight(pRect));
  InvalPortRect(&pRect);
}
```

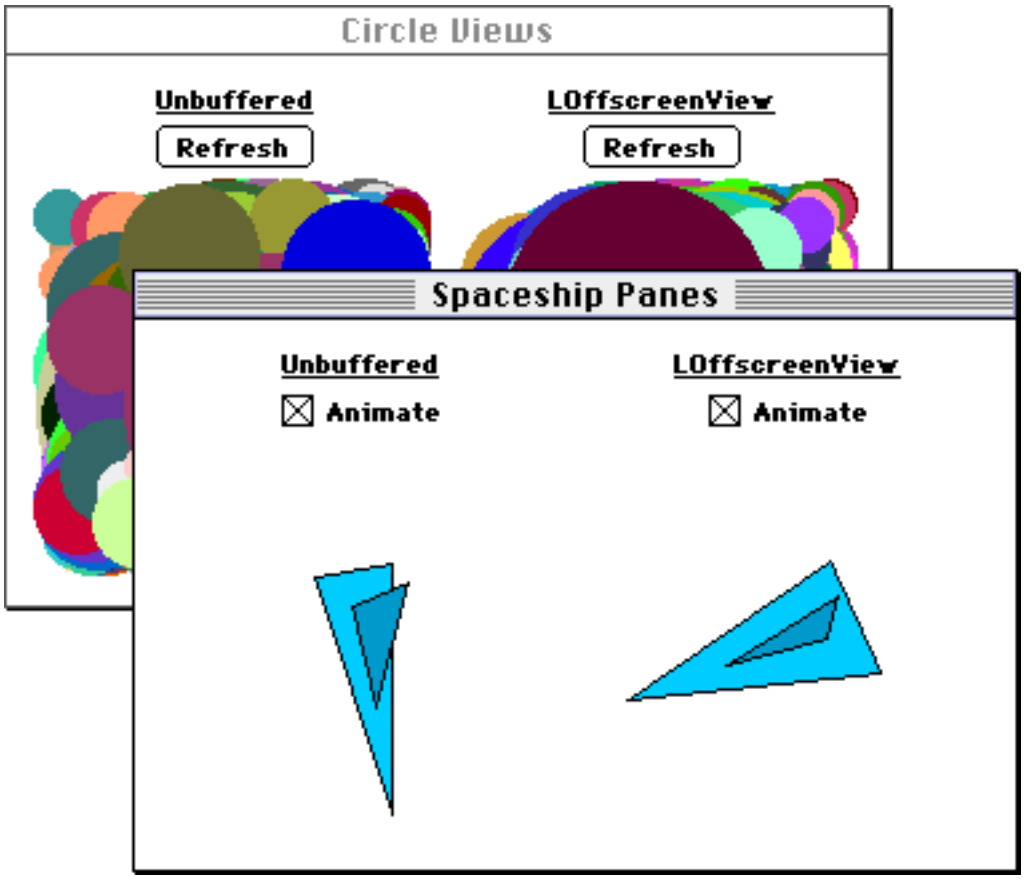
The `AddRectToImage()` function draws a rectangle to the offscreen image and forces an update of the newly drawn area. The `DrawSelf()` function just copies the image from the offscreen image into the frame of the pane. All the rectangles painted by calling `AddRectToImage()` are accumulated in the offscreen GWorld, and are redrawn properly on any subsequent screen update.

WARNING! You should be aware that LGWorld has limitations with respect to updating GWorlds, and subclassing. See “LGWorld Limitations” on page 318.

Code Exercise for Offscreen Drawing

In this exercise you create an application that demonstrates the visual difference between drawing to screen and drawing offscreen. The finished application looks like Figure 9.1.

Figure 9.1 The Offscreen application



The Circle Views window shows two groups of random circles. This window demonstrates the effect of offscreen drawing on updating. There are buttons you can use to force an update of either view.

The Spaceship Panes window shows two polygonal spaceships. Each spaceship rotates. There are buttons to turn the animation on or off for each pane. This window demonstrates the effect of off-screen drawing on animation.

In this exercise you implement the offscreen drawing sections of each window. This is a very unusual *code* exercise, because you don't write any code! All your work will be in Constructor.

1. Build and run the Offscreen application.

`project file` `Offscreen Start Code folder`

Before doing anything else, build and run the start code application. The principal goal of this step is to show you how the application uses regular drawing, and to highlight the fact that after you are done with Constructor, the same code works for offscreen drawing.

Open either the `Offscreen.68K.μ` or the `Offscreen.PPC.μ` project file. Make sure you use the project in the `Offscreen Start Code` folder, not the solution code. Then, *without making any changes*, build and run the application. When it builds successfully, you'll see the Circle Views window and the Spaceship Panes window. However, the offscreen portion of each window will be empty.

Click the refresh button above the circles in the Circle Views window. The circles underneath will redraw. Watch the drawing process carefully. You should see a series of random circles draw on top of each other. There are, in fact, 500 circles. The speed of drawing will vary depending upon the speed of the computer running the application. On fast machines, the circles draw quickly. So watch closely.

NOTE: The constant `kNumCirclePanes` controls the number of circles. It is defined in CCircleView.h if you wish to change it.

Also observe the rotating spaceship in the Spaceship Panes window. Notice how it flashes or blinks as it rotates. You can click the check box to turn animation off or on as you wish.

Both of these conditions (circles drawing on top of each other and the flashing spaceship) are artifacts of drawing direct to screen. In the next few steps you set up the application to draw the exact same items through an offscreen buffer.

For now, quit the Offscreen application.

2. Create an offscreen circle view

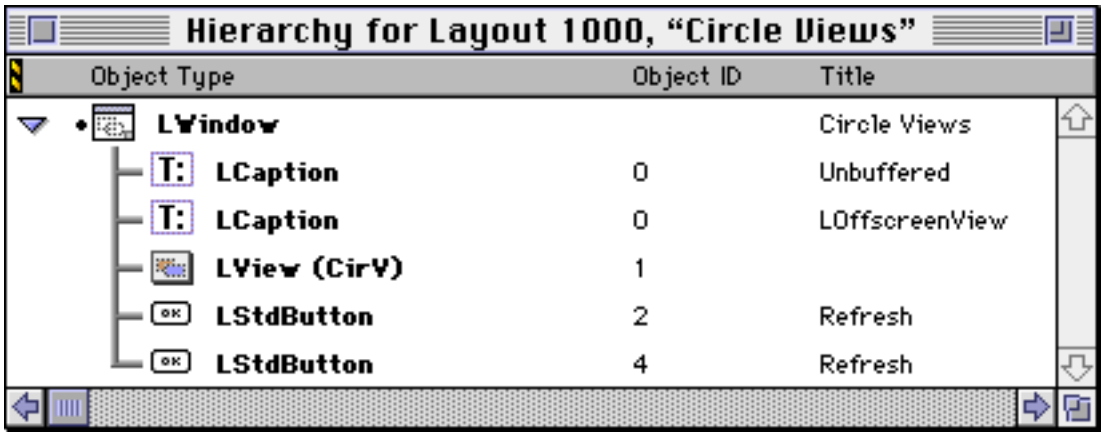
`Circle Views (ID 1000)` `Offscreen.ppob`

In this step you duplicate the existing circle view, and place it inside an LOffscreenView object. As a result, the new view will draw offscreen rather than directly to screen.

Double-click the `Offscreen.ppob` file in the CodeWarrior project manager window. This opens the resource file in Constructor.

In the Constructor project window, double-click the Circle Views resource (in the Windows and Views) to open up that resource. Finally, open a hierarchy window (**Show Object Hierarchy** in the **Layout menu**). Figure 9.2 shows what you should see.

Figure 9.2 The Circle Views hierarchy window



To complete this step you should perform the following tasks.

- Add an LOffscreenView. Add this at the same level as the other items in the window. Make it the same size as the existing circle view—LView (CirV).
- Make a copy of the existing circle view.
- Place the copy inside the LOffscreenView.
- Set the new circle view properties. Position the top left of the new LView at location (0,0). Set the pane ID to 3.

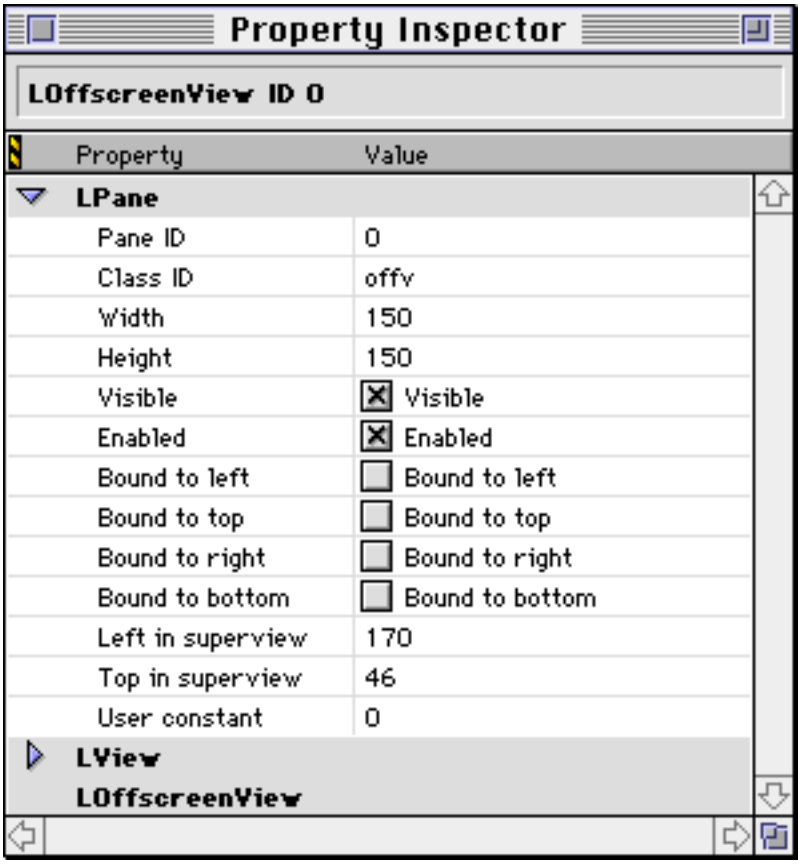
If you are comfortable with Constructor, just perform these tasks. If you want additional guidance, read the detailed substeps.

a. Add an LOffscreenView.

Choose **Catalog** from the **Window** menu. Drag an LOffscreen-View from the Catalog window into either the hierarchy window or the PPob window.

Open the inspector window for the new LOffscreenView. Set the top to 46, the left to 170, the width and height to 150. All other values are default. Figure 9.3 shows the end result.

Figure 9.3 The LOffscreenView properties



b. Make a copy of the existing circle view.

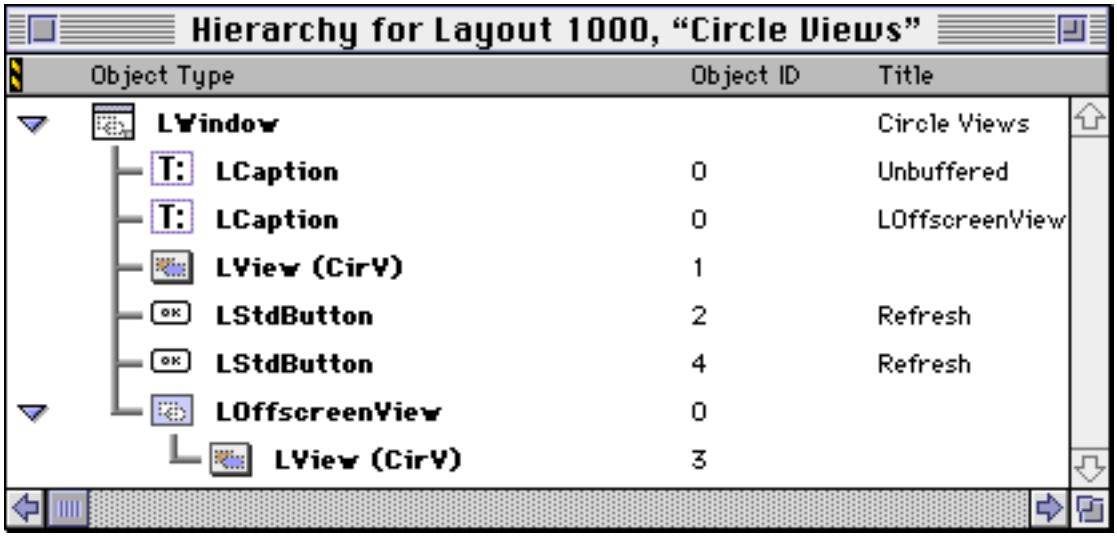
c. Place the copy inside the LOffscreenView.

You can accomplish both these substeps with a single gesture in the hierarchy window by Option-dragging to copy the item.

In the hierarchy window, select the existing circle view object—the LView (CirV) object. Then press the Option key and drag the circle view underneath the new LOffscreenView. Drop it there.

When you are through, the hierarchy window should look like Figure 9.4. Note especially the position of the new circle view underneath and indented one position to the right of the new LOffscreenView. This tells you that the new circle view is hierarchically inside the LOffscreenView.

Figure 9.4 The hierarchy window after copying LView



d. Set the new circle view properties.

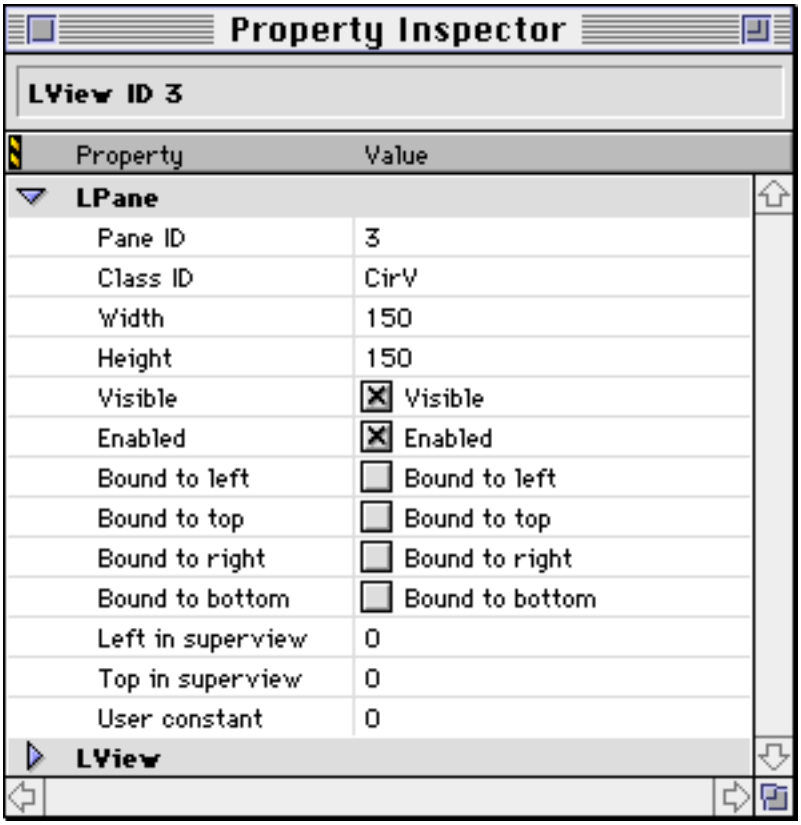
You must set the location of the new circle view within the LOffscreenView, as well as the pane ID for the new circle view object.

Open the inspector for the new circle view. To do this, double-click the new LView (CirV) in the hierarchy window. Set the top

left corner to (0,0). This puts the circle view at the same spot as the top left corner of the LOffscreenView.

Also, set the pane ID to 3. The application looks for this object by pane ID, and that ID must be 3. When you are finished, the properties for the new circle view should look like Figure 9.5.

Figure 9.5 The new circle view properties



You have completed this step. You have duplicated the existing circle view, and placed it inside an offscreen view. Note that the circle view you created is functionally *identical* to the original view that draws on screen. The only differences are for PowerPlant housekeeping. You changed the location and pane ID of the view.

At runtime the exact same code for the circle view class will run for both the direct-to-screen and offscreen drawing.

3. Create an offscreen spaceship pane

`Spaceship Panes (ID 1100)` `Offscreen.ppob`

In this step you duplicate the existing spaceship pane, and place it inside an LOffscreenView object. As a result, the new pane will draw offscreen rather than directly to the screen.

This step is essentially identical to the step you just completed. The tasks and substeps are the same, except that you're working on the spaceship pane instead of the circle view.

To complete this step you should perform the following tasks.

- Add an LOffscreenView. Add this at the same level as the other items in the spaceship window. Make it the same size as the existing spaceship pane.
- Make a copy of the existing spaceship pane.
- Place the copy inside the LOffscreenView.
- Set the new spaceship pane properties. Position the top left of the new spaceship pane at location (0,0). Set the pane ID to 3.

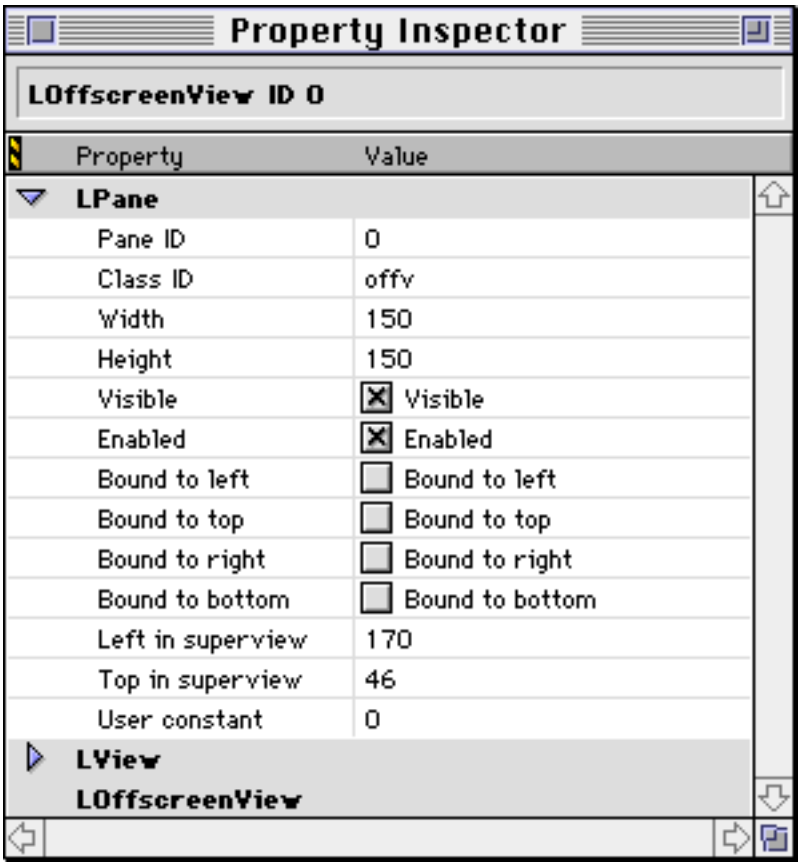
If you are comfortable with Constructor, just perform these tasks. If you want additional guidance, read the detailed substeps below.

a. Add an LOffscreenView.

Choose **Catalog** from the **Window** menu. Drag an LOffscreenView from the Catalog window into either the hierarchy window or the PPob window for the spaceship pane.

Then open the inspector window for the new LOffscreenView and set its properties. Set the top to 46, the left to 170, and the width and height to 150. All other values are default. Figure 9.6 shows the end result.

Figure 9.6 The LOffscreenView properties



b. Make a copy of the existing spaceship pane.

c. Place the copy inside the LOffscreenView.

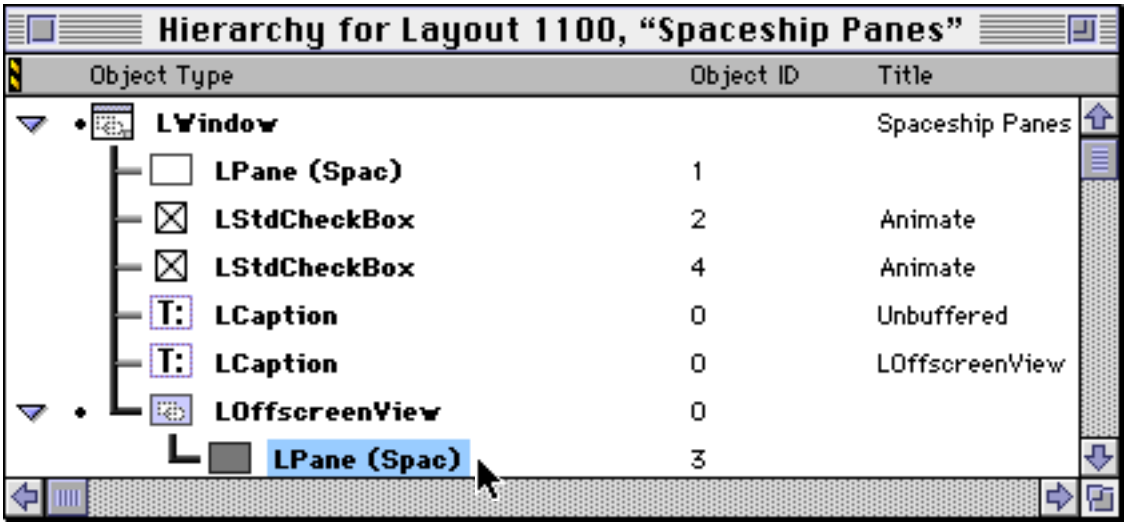
You can accomplish both these substeps with a single gesture in the hierarchy window by Option-dragging to copy the item.

In the hierarchy window, select the existing spaceship pane object—the LPane (Spac) object. Then press the Option key and drag the spaceship pane object underneath the new LOffscreenView. Drop it there.

When you are through, the hierarchy window should look like Figure 9.7. Note especially the position of the new spaceship pane underneath and indented one position to the right of the

new LOffscreenView. This tells you that the new spaceship pane is hierarchically inside the LOffscreenView.

Figure 9.7 The hierarchy window after copying LView

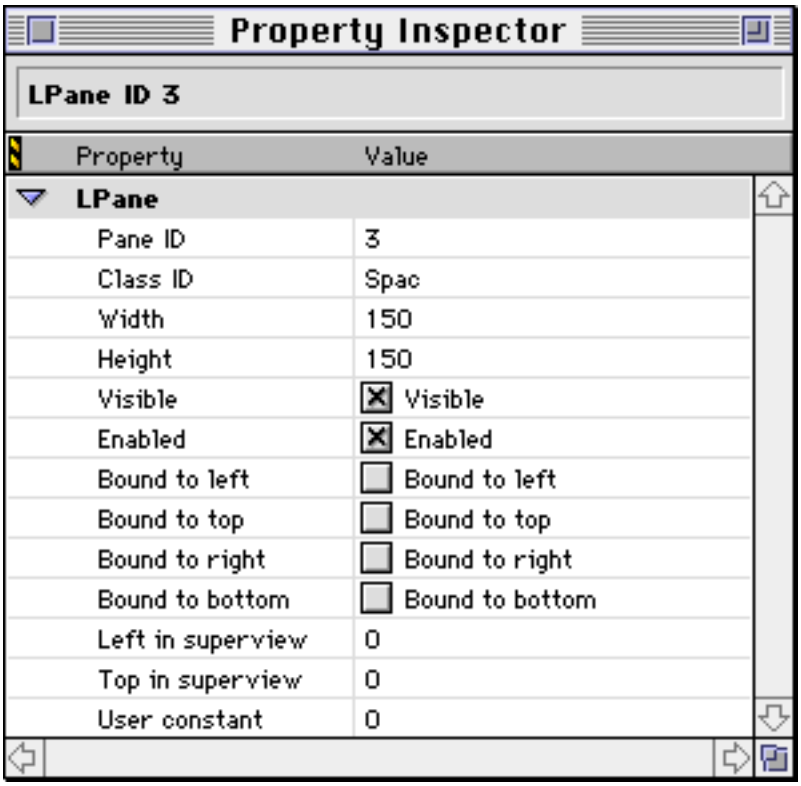


d. Set the new spaceship pane properties.

You must set the location of the new spaceship pane within the LOffscreenView, as well as the pane ID.

Open the inspector for the new spaceship pane. Set the top left corner to (0,0). Set the pane ID to 3.

Figure 9.8 The new spaceship pane



You have duplicated the existing spaceship pane, and placed it inside an offscreen view. Note that the spaceship pane you created is functionally *identical* to the original pane that draws on screen. The only differences are for PowerPlant housekeeping. You changed the location and pane ID of the new spaceship pane.

At runtime the exact same code for the spaceship pane class will run for both the onscreen and offscreen versions.

Save your changes, close the windows, and quit Constructor. You're all done.

4. Build and run the application.

Switch back to the CodeWarrior IDE, and run the application. If you ran the application as instructed in Step 1, all the code has already been compiled. Note that none of the code has to be recompiled. In

the build process, the only thing that changes is that the new PPob resource file is copied into the application.

When the project builds and runs successfully, both the Circle Views window and the Spaceship Panes window appear.

In the Circle Views window, click the Refresh button above each circle view. Observe the difference in appearance. As the screen circle view updates, each of the 500 circles is visible as it draws, despite the fact that hundreds of them are ultimately concealed.

As the offscreen circle view updates, there is a brief pause, and then the finished drawing appears all at once.

Refresh each circle view several times to get a feel for the perceptual difference between the two techniques. How does it look to you? Which one better represents what you are trying to show to a user? Offscreen drawing hides intermediate drawing steps from the user. This is particularly useful when drawing must overlap. Offscreen drawing can give you a significant enhancement to the look and feel of your application.

Now take a look at the spaceships. When the application starts, both spaceships should be animated. Observe the visual difference between the two techniques: direct-to-screen and offscreen. Once again, how do they look? Which one presents a better experience to the user? Feel free to turn animation off and on at will.

Although less obvious than the random circle views, overlapping drawing is the cause of flicker in the direct-to-screen spaceship pane. The flicker is the result of a repeated cycle of erasing and redrawing. With the offscreen view, only the final result of the erase/draw combination appears on screen. The actual process of erasing and drawing is hidden from the user.

Congratulations! You have implemented offscreen drawing, and you didn't even write any code! Examine the source code for the Offscreen application if you have any doubts about how the circle view and spaceship pane draw.

Neither makes any assumption about where it is drawing. Each simply draws itself like any ordinary pane or view. The real difference is in Constructor, where in one case the pane is placed directly in a window, and in the other it is placed inside LOffscreenView.

COffscreenApp inherits from LListener (to listen to the controls in the windows) and LPeriodical (to animate the spaceships).

Examine the COffscreenApp constructor. It does what any ordinary application constructor does. It registers the necessary classes and creates windows. It also sets itself up as an idler so it can animate the spaceships. There is no offscreen magic going on here.

Each circle view adds `kNumCirclePanes` to itself. See `CCircleView::FinishCreateSelf()` for a good example of building panes on the fly. Each random circle pane simply paints a circle.

Examine the `ListenToMessage()` function in COffscreenApp. All it does is tell the appropriate circle view to refresh itself. This causes an update event. When the application receives the update event, the view draws itself. This is when—for the offscreen drawing—LOffscreenView performs its magic. You should examine the `LOffscreenView::Draw()` function if you have any questions.

Examine the `SpendTime()` function in COffscreenApp. This is where the spaceships are animated. After rotating each ship, the function calls `UpdatePort()` to redraw the entire spaceship window. It could wait for an update event (generated by the rotation calls), but this would result in animation that might skip frames and appear uneven. Instead, `SpendTime()` draws immediately for smoother animation.

Once again, the code is executing the standard PowerPlant drawing mechanism. It is the presence of LOffscreenView in the visual hierarchy that makes all the difference.

For further exploration, you might want to experiment with using LGWorld for the offscreen spaceship pane (or the circle view for that matter). See “Using LGWorld” on page 324 for some tips. The effect on the circle view would probably be more noticeable. Why? Because you don’t have to redraw all 500 circles every time there’s an update. The circles don’t move or change. You can simply blit the LGWorld offscreen bitmap to the screen. You should see a significant speed improvement.

Finally, if you're interested in 3D graphics, you might want to take a look at the functions in the U3DDrawing class. These are very simple functions for rendering polygons.

There are many graphics resources available online. The Usenet newsgroup `comp.graphics.algorithms` has a good FAQ.

For hidden surface information, there is an FAQ and source code for Binary Space Partitioning (BSP) Trees at:

`<http://www.qualia.com/~bwade/>`

This site has a nice sample application done with CodeWarrior that includes excellent C++ classes for 3D graphics. The sample demonstrates a 3D rotation controller called "arcball." The spaceship shape in this example is based on the original demonstration of arcball from:

`<ftp://ftp.cis.upenn.edu/pub/graphics/arcball/>`

10

Profiling in PowerPlant

The PowerPlant Profiler Class is a wrapper class for the Profiler API. It provides an easy to use interface for the initialization and termination functions of the profiler.

Introduction to Profiling in PowerPlant

If you have used the profiler before to profile C or Pascal code, you are already familiar with how the initialization and termination routines work. In PowerPlant, you never need to call these routines directly. Instead, these calls are handled for you by the constructor and destructor of the profiling class.

In this chapter, you'll learn:

- Profiling Strategy—a brief review of profiling in general
- Profiling Class—shows you how the profiling class is set up and what is required to use it
- Profiling Your Code—a short description on how to profile your code

There is a code exercise provided at the end of this chapter to give you a full working tutorial on how to use the profiler class in your PowerPlant programs.

Profiling Strategy

Before you delve into the depths of the Profiler Class, you should familiarize yourself with the profiler in general. We recommend that you read the *Profiler Manual* first to become familiar with all aspects

of the profiler. Some information is reiterated here, but you should read the *Profiler Manual* for more in-depth information.

You use a profiler to measure the runtime performance of your code. What is usually important is how your code's performance measures up to some standard. When approaching the problem of measuring performance, you might want to take these three steps.

1. **Establish your standards.**

   For example, you might decide that you want the program to load in less than ten seconds, or check the spelling of a five-page document that contains no misspellings in 15 seconds. Decide the platform you will use for testing as well: a Power Macintosh is a bit faster than a Mac Plus.

2. **Determine how to measure time.**

   Your measurement device may be no more complicated than a stopwatch, or you may need to add some simple code to count ticks. At this phase you want to test the code in as close to its finished form as possible, so measure time in a way that is accurate enough to suit your needs, and that has the lowest impact on your code's natural performance. You do not want to run a full-blown profile here, because profiling can add significant overhead, thus slowing down your code's raw performance.

3. **Run the tests and measure results.**

   If you meet your performance goals, your job is done. If your code does not meet your goals, then it's time to profile your code.

Rules of Thumb

To make sure you get a good profile of your code, there are few general rules of thumb you need to follow.

1. **Turn off all optimizations**

   Optimization settings will cause skewed results in the profiler output. Part of the reason for profiling your code is so you can optimize it yourself as much as possible. Once you are confident you've done

all you can, you can turn on optimizations in the final build of your application.

2. **Turn off virtual memory or RAM Doubler**

   Profiling code with virtual memory or RAM Doubler active produces erratic and skewed result times. If your program runs quickly without virtual memory or RAM Doubler active, it will run quickly with them active.

3. **Disable Speed Doubler**

   Speed Doubler is a performance enhancement series of extensions. As such, it will skew your result times and should therefore be disabled while profiling your code.

TIP: It's actually best to turn off all extensions when doing a profile of your code. This will eliminate any chances of extension problems and background tasks skewing the profile results. Restart your computer with the Shift key held down to disable extensions.

4. **Do not use `UnloadSeg()` in 68K code**

   Calls to `UnloadSeg()` will cause garbage output when you look at your data with the MW Profiler application. Comment out, or remove calls to `UnloadSeg()` until the final build of your program.

Profiling Class

The PowerPlant profiler class is called `StProfileSection` and is found in `UProfiler.cp`. The "St" means that this is a stack-based class and functions accordingly. This class is an independent class and can therefore be used with or without the rest of PowerPlant.

`StProfileSection` makes two assumptions when calling `ProfilerInit()`. First, it uses `collectDetailed` as the data collection method. Second, it uses `bestTimeBase` for the timing method parameter. `StProfileSection` only has two member functions, a constructor and a destructor.

See Also the *Profiler Manual* for more information on the parameters used for `ProfilerInit()`.

StProfileSection

The class constructor takes three parameters and performs two main operations. It initializes three data members with the values you provide, and calls `ProfilerInit()` using these values. The constructor is declared as follows:

```
StProfileSection( Str255 inDumpFileName,
    Int16 inNumFunctions,
    Int16 inStackDepth);
```

TIP: If you are profiling your entire PowerPlant program, you should use large values for `inNumFunctions` and `inStackDepth`. The values you use will depend on the size of your program. For example, use 2500 for `inNumFunctions` and 100 for `inStackDepth` to start, then increase either of these values if necessary after each profile run. Also, remember it may become necessary to increase the memory size of your application program as profiling takes more memory than would normally be required.

The three data members are shown in Table 10.1.

Table 10.1 StProfileSection data members

Data Member	Stores
`mProfilerDumpFile`	pascal string for the dump filename
`mNumFunctions`	number of routines to create buffer space for
`mStackDepth`	number of routines deep the stack can get

~StProfileSection

`~StProfileSection()` is the class destructor. It is called automatically as soon as the code you are profiling goes out of scope. If you are profiling one area of your code, it's when that function ends. If you are profiling your entire application, it's when the application quits.

The destructor does all the house cleaning work. It calls `ProfilerSetStatus()` to stop profiling while it performs the clean up, calls `ProfilerGetDataSizes()` to test memory requirements and sends an `Assert_()` if there wasn't enough memory, calls `ProfilerDump()` to dump the results to the file, and finally, calls `ProfilerTerm()` to end the profile session.

NOTE: If you get an `Assert_()` message, it means the values you provided for `inNumFunctions` and `inStackDepth` are too low. You will need to re-profile your code using larger values.

Profiling Your Code

To profile your code, you do four things.

1. **Set up your project for profiling**

   This step is very similar to the description given in the *Profiler Manual*. The only changes are the source file you add to your project, and the header file you add to your code. More detail on this is given in the code exercise.

2. **Run a profiler on the area of the code you want tested.**

   This might be a single routine, a group of routines that perform a task, or even the entire application. What you profile depends upon what you are testing.

   Declare a local `StProfileSection` variable, and provide the proper setup information. Your declaration will look something like this:

```
StProfileSection theProfile("\pTest", 50, 50);
```

All code that runs from that point on, until `theProfile` goes out of scope, will be profiled.

3. **Analyze the data collected by the profiler and improve your code.**

   You study the results of your profiling and look for problems and room for improvement. For details on how to review the results, see the *Profiler Manual*.

   The profiling process is iterative. You repeat steps 2 and 3 until you achieve the performance gain you need to meet your goals.

4. **Run your code without the profiler**

   When you are satisfied that you have reached your goals, you have one more step to perform. You should run your original tests—without the profiler of course—to verify that your code in its natural state meets your performance goals.

   Implementing the above steps in your program is simple and only requires two lines of code.

Code Exercise for Profiling

This code exercise uses the Documents *solution* code from the File I/O chapter of *The PowerPlant Book*. The code can be found on the CodeWarrior Reference CD, in the `CodeWarrior Examples: MacOS Examples:PP Book Code:Chap 13 Solution Code` folder.

WARNING! It is important for this exercise that you use the **Solution** code for the Documents application. Using the Starter Code will either not compile, or crash when run, unless you have finished it in the exercise in The PowerPlant Book.

In this exercise you will perform two different profiles. The first profile will only focus on one routine (and any routines it calls). The second profile will be the entire application. The steps for each are similar.

Profiling a Single Routine

There is one important thing to remember when you want to profile a single routine. You are also profiling all other routines called directly or indirectly by the routine being profiled. This is important in order to provide the correct parameters to the constructor. This part of the code exercise will profile the `OpenFile()` routine in `CTextDocument.cp`.

WARNING! When you are profiling a single routine, you must declare your `StProfileSection` variable in the function that calls the routine you want to profile. If you declare an `StProfileSection` variable inside the function you want to profile, you will not see the name of your routine in the MW Profiler application. Your routine would still be profiled, but you would be missing an important point of reference in the resulting data.

1. Set up the project for profiling

There are two things you need to do for this step. You can do these in any order you wish. First, add the necessary source and library files to your project. Then turn on profiling from the preference panel. These steps are the same for both 68K and PowerPC projects.

a. Adding files to the project

Add `UProfiler.cp` to your project as well as the appropriate library. The library you add depends on the type of program you are writing and your target platform (68K or PowerPC). Table 10.2 lists the libraries and shows you when to use them.

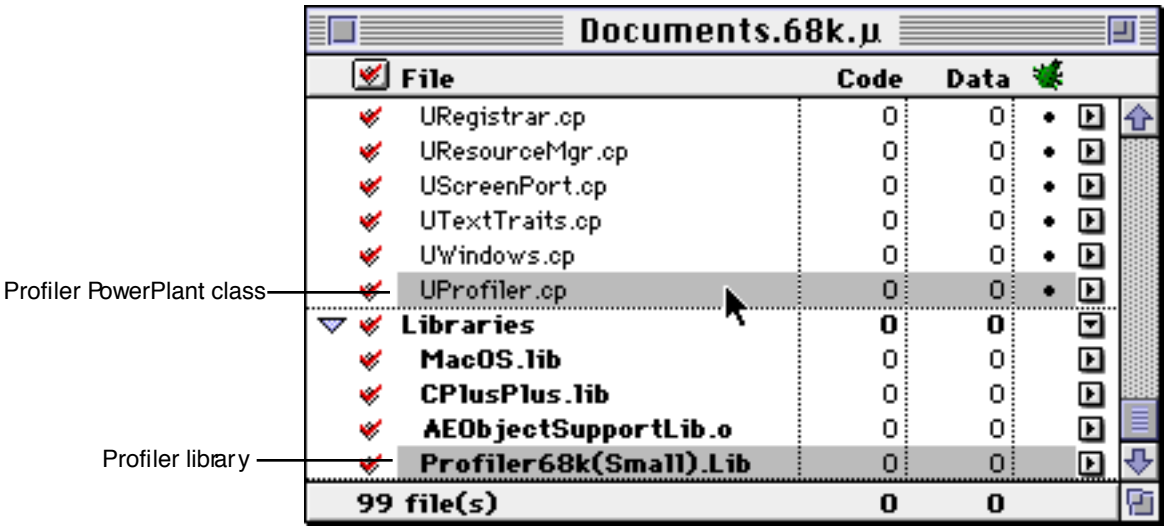
Table 10.2 Profiler libraries

Library	Processor	When To Use
`profiler68k(Small).lib`	68K (A5 globals)	near/small model applications
`profiler68k(Large).lib`	68K (A5 globals)	far/large model applications
`profiler68kA4(Small).lib`	68K (A4 globals)	near/small model code resources
`profiler68kA4(Large).lib`	68K (A4 globals)	far/large model code resources
`Profiler68kCFM.lib`	68K (A5 globals)	code fragments and shared libraries
`ProfilerLIB`	CFM 68K & PowerPC	code fragments and shared libraries
`profilerPPC.lib`	PowerPC	PowerPC applications and shared libraries

The example shown in Figure 10.1 uses `Profiler68K(Small).Lib` for this exercise.

See Also the *Profiler Manual* for more information on profiler libraries.

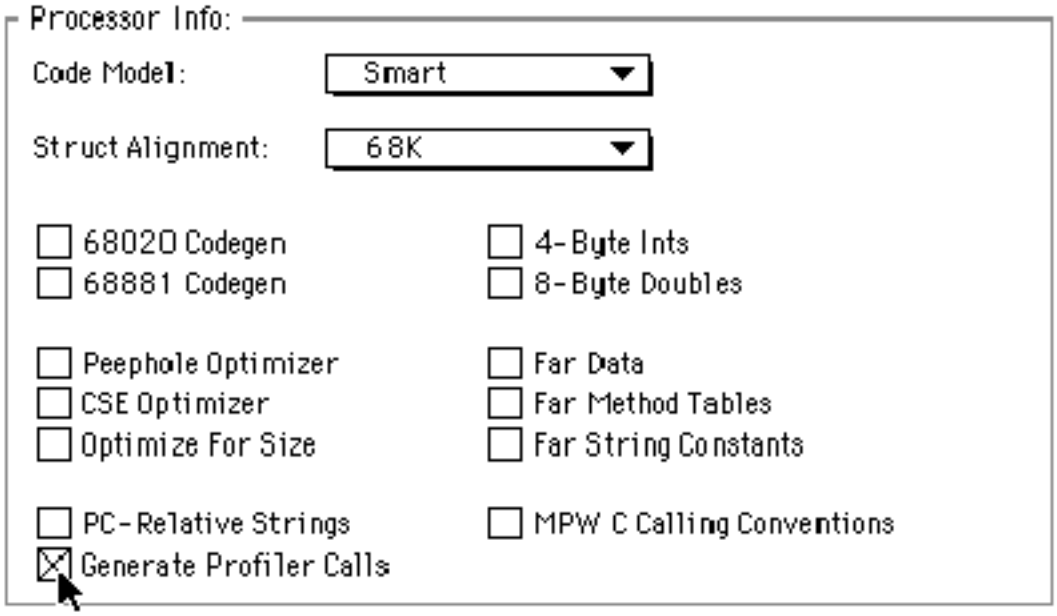
Figure 10.1 Adding profiler files to the project



b. Turn on profiling for the project

Select **Generate Profiler Calls** from the **68K Processor** preference panel as shown in Figure 10.2.

Figure 10.2 Generate profiler calls



NOTE: This option is called **Emit Profiler Calls** in the **PPC Processor** preference panel.

2. Add the header file

`CTextDocument.cp`

You need to include `UProfiler.h` in the file that makes the call to the `StProfileSection` class. It is not needed in any other file in your project even if the routine your profiling calls routines in a different file.

```
#include <UProfiler.h>
```

3. Declare an `StProfileSection` variable

`CTextDocument()` `CTextDocument.cp`

In order to profile the routine you want, you declare the `StProfileSection` variable in the routine that makes the call to the func-

tion you are concerned with. In this case, you put the declaration in the class constructor right before the call to `OpenFile()`.

```
// Set name of window or open file.
if ( inFileSpec == nil ) {

  NameNewDoc();

} else {
  // Profile the OpenFile routine
  StProfileSection theProfile("\pTest", 50, 50);
  OpenFile( *inFileSpec );

}
```

4. Build and run the application

That's all there is to it! Once the application builds successfully, run the application, open a file, then quit. There will be a file called "Test" in the same folder as the application. You can then use the MW Profiler application to view the results.

See Also the *Profiler Manual* for more information on how to view your data.

Profiling an Application

You might decide you want to get the "big picture" of what's happening in your code. To do that, you would profile your entire PowerPlant program.

The steps you perform are exactly the same as shown above with the exception of steps two and three. If you've performed the above tasks, remove the header file and the declaration of `StProfile-Section` and replace them with the following steps.

1. Add the header file

`CDocumentsApp.cp`

Add the header file to the top of `CDocumentsApp.cp`.

```
#include<UProfiler.h>
```

2. Declare the StProfileSection variable

`main()` `CDocumentsApp.cp`

You do not necessarily need to profile the initialization code unless you feel it is a concern. The code below declares the class before the `Run()` command is given. You may put it at the top of `main()` if you wish.

```
// Create the application object and run it.
CDocumentsApp theApp;
StProfileSection theProfile("\pApp", 2500, 100);
theApp.Run();
```

3. Build and run the application

That's all there is to it! Once the application builds successfully and runs, a complete profile of the application will be built. You have to quit the application for the `StProfileSection` class to go out of scope, write the data, and close the file. Once you've done that, there will be a file called "App" in the same folder as the application. You can then use the MW Profiler application to view the results.

TIP: If you run multiple profiles of your code, the profiler routines are smart enough not to overwrite the same dump file. Instead, it will append a number after the file name (e.g.: App2). This is useful if you want to compare different profiles of your program (as you make changes) and not have to change your code. For example, you may want to compare the 68K version with the PowerPC version. For reliable comparisons, make sure you profile exactly the same routines each time, and in each version.

See Also the *Profiler Manual* for more information on how to view your data.

Index

Symbols

~LGWorld() destructor 316
~StOffscreenGWorld() destructor 320
~StProfileSection() 343

A

AcceptIncoming()
  and accepting connection 241
  and responder 239
  LEndpoint 228
AckSends()
  LEndpoint 230, 233
actions
  class hierarchy 99
  creating classes 104
  described 98
  in LCaption class 98
  posting 99, 105
  see also undo
ActivateSelf()
  LTableView 128
AddDesc()
  UAEDesc 184
AddDropArea()
  LDropArea 21
AddFlavors()
  LDragTask 19, 20, 30
AddKeyDesc()
  UAEDesc 184
AddLastChildNode()
  LCollapsableTree 140
AddLastChildRow()
  LHierarchyTable 132
AddPtr()
  UAEDesc 184
AddRectDragItem()
  LDragTask 19, 20, 30
address, obtaining network 236
AddSubModel()
  LModelObject 176
aedt resource 188
AEOM 169
  and aete resource 172, 186
  and LModelObject 172
  and LModelProperty 180
  class stored in mModelKind 174
  concepts 169
  containment hierarchy 174
  terminology 170
aete resource 186
  editing 187
  editing with Resorcerer 188
AllocateThreads()
  LThread 53, 65
allocating memory for threads 65, 66
Apple event
  adding an AEOM class to your code 189
  adding custom events 191
  adding properties to your AEOM class 191
  classes 171–185
  default submodel 193
  described 169
  implementing 188–194
  lazy objects 192
  recordability 194
  resources 186–188
  set tell target 193
  strategy 169–170
  whose clause 194
Apple Event Object Model See AEOM
asynchronous behavior
  in threads classes 81–87
asynchronous threads
  Speech Manager example 86
attaching an undoer 105

B

background highlighting 25
BeginDrawing()
  LGWorld 316, 317
bestTimeBase 341
Bind()
  and IP address 237
  and listening for connection 240
  and responder class 239
  LEndpoint 228, 232
Block()

LThread 48, 54
blocked thread 48

C

callback functions in drag and drop 33
CanRedo()
LAction 101
CanUndo()
LAction 101
cell
addressing 118
as STableCell object 119
changing size 146
data storage size 139
drawing 149
finding 151
finding data in 152
finding next selected 138
getting data 147
handling click in 148
highighting 150
location in PowerPlant 123
nesting level 150
order in table 118, 151
refreshing 151
see also table
setting data 146
CellIsSelected()
LTableSelector 136
LTableView 127
CellToIndex()
LTableView 125
CheckForMissingParameters()
UAppleEventMgr 183
click
in drag and drop 25
in table cell 148
Click()
overriding LPane for drag and drop 25, 26
ClickCell()
LTableView 129
overriding 149
ClickInContent()
LWindow 28
ClickSelect()
LTableSelector 136
LTableView 127
ClickSelf()
LHierarchyTable 148
LTableView 128, 148
client
compared to responder 240
creating 235
creating endpoint 236
described 234
client/server
described 234
coerce data in Apple event 181
collectDetailed 341
Connect()
LEndpoint 228
context switching in threads 72
CopyImage()
LGWorld 316, 317
CountReadyThreads()
LThread 53
CountSubModels()
LModelObject 176
CreateInternetMapper()
UNetworkFactory 224
CreateTCPEndpoint()
creating endpoint 241
UNetworkFactory 224
creating threads 65
criticality 71
current drop area 21
current thread 47
custom callbacks for drag and drop 32

D

data coherency in threads 71–81
datagram 243
DeactivateSelf()
LTableView 128
default submodel 193
delaySelect
attribute in drag and drop 26
DeleteThread()
LThread 55, 56, 70
deleting threads 57, 69
Disconnect()

LEndpoint 228
DoDrag()
LDragTask 19, 20, 30
DoDragDrawing()
LDropArea 33
DoDragInput()
LDropArea 33
DoDragReceive()
LDropArea 22, 23, 32
DoDragSendData()
LDropArea 33
DoForEach()
LQueue 62
LThread 53
DontAckSends()
LEndpoint 230, 233
double buffering See offscreen drawing
drag and drop
adding flavors 20, 30
adding to a pane 23
background highlighting 25
class hierarchy 18
classes 18–24
clicking in background window 25
creating a drag task 28
current drop area 21
custom callback functions 33
custom callbacks 32
delaySelect attribute 26
detecting a drag 28
finding Drag Manager 21, 24
handling click 25
highlighting drop area 31
implementing 24–34
implementing special behavior 31
initiating drag 20, 30
item acceptability 31
multiple items or flavors 29
promise 33
receiving a drop 22, 32
simple technique 29
strategy 16–18
tracking a drag 31
Drag Manager
finding 21, 24
drag region 19, 20, 29, 30
drag task 28
DragAndDropIsPresent()
LDropArea 21, 24
DragIsAcceptable()
LDropArea 22
DragLib, importing weak 25
DragSelect()
LTableSelector 136
Draw()
LDropFlag 141
LOffscreenView 322
DrawCell()
LSmallIconTable 130
LTableView 128
LTextHierTable 133
overriding 149
DrawDropFlag()
LHierarchicalTable 150
DrawSelf()
LTableView 128
drop flag
drawing for table row 150
including icons 141

E

EndDrawing()
LGWorld 316, 317
endpoint See LEndpoint
EnterCritical()
LThread 54, 71
EnterDropArea()
LDropArea 22, 31
ExecuteSelf()
LUndoer 102
LYieldAttachment 59
ExitCritical()
LThread 54, 71
expansion triangle in table See drop flag

F

Finalize()
in multilevel undo 106
LAction 101
FindCellData()
LTableStorage 138
LTableView 127

parameters 152
search algorithm 152
FindDropArea()
LDropArea 21
FindUndoStatus()
LUndoer 102
FinishCreateSelf()
to attach helpers in table 143
flavors
adding to a drag 20, 30
FocusDropArea()
LDragAndDrop 24
LDropArea 22, 23

G

generic network interface 221
GetAEProperty()
LModelObject 177
GetAmountUnread()
LEndpoint 230
GetCellData()
LTableStorage 138
LTableView 127
parameters 147
uses wide-open cell 128
GetCellHitBy()
LTableView 129
GetClickCount()
in table cell 149
GetColHitBy()
LTableGeometry 134
GetColWidth()
LTableGeometry 134
LTableView 126
GetCurrentThread()
LThread 53
GetDescription()
LAction 101
GetExposedIndex()
LCollapsableTree 140
LHierarchyTable 131
GetFreeThreads()
LThread 53
GetImageCellBounds()
LTableGeometry 134
LTableView 126
GetImportantAEProperties()
LModelObject 178, 179
GetLink()
LLink 61
GetLocalAddress()
LEndpoint 229, 232
GetLocalCellRect()
LTableView 126
GetMacGWorld()
LGWorld 316
StOffscreenGWorld 320
GetMainThread()
LThread 53
GetModelKind()
LModelObject 175
GetNestingLevel()
LCollapsableTree 140, 150
GetNextCell()
LTableView 126, 151
GetNextSelectedCell()
LTableView 126, 151
GetOptionalParamDesc()
StAEDescriptor 182, 183
GetParamDesc()
StAEDescriptor 182, 183
GetParentIndex()
LCollapsableTree 140
GetPositionOfSubModel()
LModelObject 175, 176, 177
GetRemoteHostAddress()
LEndpoint 229, 232
GetResult()
LThread 55, 56
GetRowHeight()
LTableGeometry 134
LTableView 126
GetRowHitBy()
LTableGeometry 134
GetSize()
LQueue 62
GetState()
LEndpoint 230, 233
GetStorageSize()
LTableStorage 138
GetSubModelByComplexKey()
whose clauses 194

GetSubModelByName()
LModelObject 176, 177
GetSubModelByPosition()
and laziness 193
LModelObject 176, 177
GetSuperModel()
LModelObject 175
GetTableDimensions()
LTableGeometry 134
GetTableSize()
LTableView 125
GetWideOpenIndex()
LCollapsableTree 140
LHierarchyTable 131
GetWideOpenTableSize()
LHierarchyTable 131

H

HandleAppleEvent()
implementing 192
LModelObject 178
HandleClone()
LModelObject 179
HandleCount()
LModelObject 179
HandleCreateElementEvent()
LModelObject 178
sample code 190
HandleDelete()
LModelObject 179
HandleDragDrawing()
LDropArea 33
HandleDragInput()
LDropArea 33
HandleDragReceive()
LDropArea 21
HandleDragSendData()
LDropArea 33
HandleDragTracking()
LDropArea 21
HandleMove()
LModelObject 178, 179
helper classes for tables 116
hierarchical table described 117
highlight drop area 31
HiliteCell()
overriding 150
HiliteCellActively()
LTableView 128
LTextHierTable 133
overriding 150, 151
HiliteCellInactively()
LTableView 128
LTextHierTable 133
overriding 150, 151
HiliteDropArea()
LDragAndDrop 24
LDropArea 22, 31
HiliteSelection()
overriding 150, 151

I

IndexToCell()
LTableView 125
initializing threads 63
InMainThread()
LThread 53
InsertChildNodes()
LCollapsableTree 140
InsertChildRows()
LHierarchyTable 132
InsertCols()
LTableGeometry 134
LTableSelector 136
LTableStorage 138
LTableView 125
parameters 143
InsertRows()
effect in hierarchical tables 144
LHierarchyTable 132
LTableGeometry 134
LTableSelector 136
LTableStorage 138
LTableView 125
InsertSibling Nodes()
LCollapsableTree 140
InsertSiblingRows()
LHierarchyTable 131
parameters 144
InsideDropArea()
LDropArea 23, 31
InstallHandlers()

LDropArea 21, 22, 34
inter-thread communication 50, 78–81
InTrackingWindow()
LDropArea 21, 22
IP address
defined 222
IsAckingSends()
LEndpoint 230, 233
IsCurrent()
LThread 54
IsDone()
LAction 101
IsEmpty()
LQueue 62
IsNullCell()
STableCell 122
IsPostable()
LAction 101
overriding in action classes 104
IsSubModelOf()
LModelObject 175
IsValidCell()
LTableView 125
IsValidCol()
LTableView 125
IsValidRow()
LTableView 125
ItemIsAcceptable()
LDropArea 23, 31

L

LAction 100
data members 100
member functions 101
relation to LUndoer and LCommander 98
subclassing 104
laziness 192
LCollapsableTree 140
member functions 140
LColumnView 129
LCommander 99
LDocument
as example of LModelObject 173
LDragAndDrop 23–24
member functions 23
LDragTask 19–20
data members 19
member functions 19
LDropArea 20–23
custom callback functions 33
data members 20
member functions 22
static functions 21
LDropFlag 141
LeaveDropArea()
LDropArea 22, 31
LEndpoint 227–??, 231–??
functions 228, 232
LEndpoint() constructor 228, 232
LEventSemaphore 60
LGWorld 315–319
data members 315
in non-PowerPlant code 319
limitations 318
member functions 316
sample code 324
using 324
LGWorld() constructor 316
LHierarchyTable 130
data members 131
member functions 131
LInternetIPAddress
and binding to port 237
and servers 240
Listen()
LEndpoint 228
responding to network connection 241
LLink 61
LModelDirector 180
LModelObject 172–179
and AEOM 170, 172
class hierarchy 173
constructor 175
containment hierarchy 174
data members 173
handling events in 178
handling properties in 177
LDocument as example 173
LWindow as example 173
managing elements in 174
LModelProperty 180

LMutexSemaphore 60
example of use 75
LNodeArrayTree 141
LNTable (obsolete class) 116
LOffscreenView 321
and Constructor 322
using 323
LQueue 61
data members 61
member functions 62
LSemaphore 59
data members 59
LSharedQueue 62
inter-thread communication 79
LSimpleThread 57–58
LSmallIconTable 130
LTableArrayStorage 139
constructors 139
LTableGeometry 133
member functions 134
LTableMonoGeometry 135
LTableMultiGeometry 135
LTableMultiSelector 137
LTableSelector 136
member functions 136
LTableSingleSelector 137
LTableStorage 138
member functions 138
LTableView 123
as example class 123
cell access functions 125
cell geometry functions 126
cell management functions 125
cell selection functions 127
data members 124
data storage functions 127
drawing and clicking functions 128
services 124
setting helper classes 124
LTEClearAction 103
LTECopyAction doesn't exist 103
LTECutAction 103
LTEPasteAction 103
LTETextAction 103
LTETypingAction 103
LTextColumn 130
LTextHierTable 133
LThread 51–57
constructor 65
data members 52
state-related functions 54
static member functions 53
vital functions 55
LThread()
LThread 55
LUndoer 102
and redo 102
as attachment to LCommander 98
member functions 102
LWindow
as example of LModelObject 173
LYieldAttachment 58

M

main thread, creating 64
MakeAppleEvent()
UAppleEventsMgr 185
MakeBooleanDesc()
UAEDesc 184
MakeDragRegion()
LDragTask 19, 20, 30
MakeInsertionLoc()
UAEDesc 184
MakeRange()
UAEDesc 184
mapper
created by UNetworkFactory 221
creating with UNetworkFactory 223
mBounds
LGWorld 315
StOffscreenGWorld 320
mMacGWorld
LGWorld 315
StOffscreenGWorld 320
mModelKind 173
modifying thread state 68
mSaveDevice
LGWorld 315
StOffscreenGWorld 320
mSavePort
LGWorld 315
StOffscreenGWorld 320

mSubModels 173
mSuperModel 173

N

nesting level 150
networking
    accepting connection 241
    binding to local port 236
    classes 223–??
    client and responder compared 240
    client described 234
    client/server described 234
    connecting to a server 237
    creating a server 239
    creating client 235
    creating client endpoint 236
    disconnect from server 238
    generic interface 221
    implementing 234–243
    obtaining an address 236
    protocol for server 239
    rejecting connection 241
    responding to connection 241
    send data 238
    server described 234
    strategy 221–??
    TCP ports 226
next of kin in threads 53, 56
Next()
    LSharedQueue 63, 79
NextGet()
    LQueue 62
    use in LSharedQueue 63
NextPut()
    LQueue 62
    LSharedQueue 79
notifier
    and UNetworkFactory functions 224

O

offscreen drawing 313–325
    classes 315–322
    implementing 322–325
    strategy 314
operator !=()
    STableCell 123
operator ==()
    STableCell 123
operator new
    use with threads 65

P

pane
    adding drag and drop 23
PointInDropArea()
    LDragAndDrop 24
    LDropArea 22, 23
port
    binding to local 236
PostAction()
    executing action 105
    LCommander 100
    LUndoer 102
    LUndoer and LCommander compared 103
PostAnAction()
    LCommander 99
posting an action 99, 105
preemptive threads 52, 66
ProfilerInit()
    used in StProfileSection 341, 342
profiling 339–344
    strategy 339
    with RAM Doubler 341
    with Speed Doubler 341
promise in drag and drop 33
protocol for network server 239

R

RAM Doubler and profiling 341
ready thread 47
Receive()
    LEndpoint 229
ReceiveDragItem()
    LDropArea 23, 32
receiving a drop 22, 32
recordable application 194
redo See undo
Redo()
    LAction 101
RedoSelf()
    in text action classes 103

LAction 101
overriding in action classes 104
RefreshCell()
LTableView 128, 151
RefreshCellRange()
LTableView 128, 151
RejectIncoming()
and rejecting connection 241
LEndpoint 229
Remove()
LQueue 62
RemoveCols()
LTableGeometry 134
LTableSelector 136
LTableStorage 138
LTableView 125
RemoveDropArea()
LDropArea 21
RemoveNode()
LCollapsableTree 140
RemoveRows()
LHierarchyTable 132
LTableGeometry 134
LTableSelector 136
LTableStorage 138
LTableView 125
parameters 145
RemoveSubModel()
LModelObject 176
Reset()
LEventSemaphore 60
Resorcerer
and aete resource 188
responder 239
compared to client 240
result
in LSimpleThread 57
of a thread 56
Resume()
LThread 48, 54
to run a thread 67
use in main thread 64
rows, insert requires wide-open index 132
Run()
effect on deleting a thread 69
LSimpleThread 57
LThread 55
not for running a thread 68
UMainThread 58
use in main thread 64
running a thread 55, 67

S

scriptability See Apple event
SelectAllCells()
LTableSelector 136
LTableView 127
SelectCell()
LTableSelector 136
LTableView 127
SelectCellBlock()
LTableMultiSelector 137
SelectionChanged()
LTableView 127, 149
semaphore
described 49
using 74–78
send data, networking 238
Send()
and sending data 238
LEndpoint 229
SendAppleEvent()
UAppleEventsMgr 185
SendAppleEventWithReply()
UAppleEventsMgr 185
SendSelfAE()
recordability 194
server
accepting connection 241
and responder class 239
choosing a protocol 239
connecting to 237
creating 239
described 234
disconnecting 238
listening for connection 240
rejecting connection 241
responding to connection 241
set tell target 193
SetAEProperty()
LModelObject 177
SetBounds()

LGWorld 316
LGWorld example 318
SetCell()
STableCell 122
SetCellData()
LTableStorage 138
LTableView 127
parameters 146
uses wide-open cell 128
SetColWidth()
LTableGeometry 134
LTableView 126
parameters 146
SetDragDrawingProc() (Mac OS) 34
SetDragInputProc() (Mac OS) 34
SetDragSendProc() (Mac OS) 34
SetLaziness
LModelObject 193
SetLink()
LLink 61
SetModelKind()
LModelObject 175
SetNextOfKin()
LThread 55, 56
SetResult()
LThread 55, 56
SetRowHeight()
LTableGeometry 134
LTableView 126
parameters 146
SetSuperModel()
LModelObject 175
SetTableGeometry()
LTableView 124
SetTableSelector()
LTableView 124
SetTableStorage()
LTableView 124
SetTellTarget()
LModelObject 193
SetUseSubModelList()
LModelObject 175
ShowDragHilite() (Mac OS) 32
Signal()
LEventSemaphore 60
LMutexSemaphore 60
LSemaphore 48, 59
using with semaphores 74
Sleep()
LThread 48, 54
sleeping thread 48
Speed Doubler and profiling 341
STableCell 122
in calls to get/set data 128
member functions 122
StAEDescriptor 182
encoding data with 183
member functions 182
state transition in threads 48
StCritical 61
advantages of 72
SThreadParamBlk 84
StMutex 60
in mutual exclusion strategy 76
StOffscreenGWorld 319–321
data members 320
member functions 320
sample code 323
using 323
StOffscreenGWorld() constructor 320
StProfileSection 341
data members 342
destructor 343
StProfileSection() constructor 342
Suspend()
LThread 48, 54
suspended thread 48
SwapContext()
calling inherited 74
LThread 55, 64, 72

T

T_LISTEN 241
table
accessing data directly 148
addressing cell 118
cell ordering 118, 151
changing row and column size 146
class architecture 116
class hierarchy 120
classes 119–141
creating 142

creating helper object 142
drawing a cell 149
drawing drop flag 150
finding data in 152
finding particular cell 151
getting cell data 147
handling click in cell 148
helper class hierarchies 121
helper classes 116
hierarchical described 117
highlighting cells 150
implementing 141–153
inserting column 143
inserting rows 144
managing rows and columns 143–146
nesting level of row 150
refresh cells 151
removing rows and columns 145
scrolling 153
see also cell
selection changed 149
setting cell data 146
setting helper classes 124
strategy 116–119
TableIndexT 118
TCP
creating endpoint 241
ports 226
terminology resource See aete resouce
TheBoolean()
UExtractFromAEDesc 181
TheEnum()
UExtractFromAEDesc 181
TheInt16()
UExtractFromAEDesc 181
TheInt32()
UExtractFromAEDesc 181
ThePoint()
UExtractFromAEDesc 181
ThePString()
UExtractFromAEDesc 181
TheRect()
UExtractFromAEDesc 181
TheRGBColor()
UExtractFromAEDesc 181
TheType()
UExtractFromAEDesc 181
Thread Manager
finding 63
ThreadAsynchronousResume()
LThread 48
ThreadBeginCritical() (Mac OS) 49, 61
ThreadEndCritical() (Mac OS) 49, 61
ThreadProc function 57
threads
allocating memory 66
allocating memory for 65
asynchronous operations 81–87
blocked 48
calling constructor 65
class hierarchy 51
classes 50–63
Context Switching 72
context switching 72
creating 65
creating main 64
critical data 71
current 47
data coherency 71–81
deleting 57, 69
finding Thread Manager 63
implementing 63–70
initializing 63
inter-thread communication 50, 78–81
modifying state 68
next of kin 53, 56
preemptive 52, 66
ready 47
result 56, 57
running 55, 67
semaphore described 49
sleeping 48
state 47–49
state transition diagram 48
strategy 46–50
suspended 48
using semaphores 74–78
waiting 48
ThreadsLib, importing weak 64
ToggleAction()
LUndoer 102
ToPoint()
STableCell 122
TrackClick()

LDropFlag 141
TrackDrag() (Mac OS) 20, 29
tracking a drag 31

U

UAEDesc 184
member functions 184
UAEGizmos 172, 193
UAppleEventsMgr 185
member functions 185
UDP 243
UExtractFromAEDesc 180
data coercion 181
member functions 181
UMainThread 58
Unbind()
LEndpoint 228, 232
undo
attaching undoer 105
classes 98–103
command messages 102
implementing 104–107
menu strings 100
multilevel 106
see also actions, LAction
strategy 98
Undo()
LAction 101
UndoSelf()
in text action classes 103
LAction 101
overriding in action classes 104
UNetworkFactory 223
described 221
functions 224
UnhiliteDropArea()
LDropArea 22, 31
UnloadSeg()
in Profiling 341
UnselectAllCells()
LTableSelector 136
LTableView 127
UnselectCell()
LTableSelector 136
LTableView 127
UpdateGWorld() (Mac OS) 318
UProfiler.cp 341
User Datagram Protocol 243

W

Wait()
LMutexSemaphore 60
LSemaphore 48, 59
using with semaphores 74
waiting thread 48
WaitMouseMoved() (Mac OS) 28
Wake()
LThread 48, 54

Y

yield with LYieldAttachment 58
Yield()
LThread 47, 53, 58

CodeWarrior

PowerPlant Advanced Topics

Credits

writing lead: Jim Trudeau

other writers & Sample Code: Harold Ekstrom
Carl Constantine—Profiling
Jeremy Roschelle—Apple events
Joe Zobkiw—Networking
Chris Haupt—Internet

engineering: Greg Dow, Eric Scouten, Frank Vernon, Jon Hess, Paul Lalonde, Doug Watkins

frontline warriors: Kevin Bell, John Daub, Ron Davis, Stephen Gilardi, Matt Henderson, Brian Stern, and Code Warriors everywhere

Guide to CodeWarrior Documentation

If you need information about...	See this
Installing updates to CodeWarrior	QuickStart Guide
Getting started using CodeWarrior	QuickStart Guide; Tutorials (Apple Guide)
Using CodeWarrior IDE (Integrated Development Environment)	IDE User's Guide
Debugging	Debugger Manual
Important last-minute information on new features and changes	Release Notes folder
Creating Macintosh and Power Macintosh software	Targeting Mac OS; Mac OS folder
Creating Microsoft Win32/x86 software	Targeting Win32; Win32/x86 folder
Creating Java software	Targeting Java Sun Java Documentation folder
Creating Magic Cap software	Targeting Magic Cap; Magic Cap folder
Using ToolServer with the CodeWarrior editor	IDE User's Guide
Controlling CodeWarrior through AppleScript	IDE User's Guide
Using CodeWarrior to program in MPW	Command Line Tools Manual
C, C++, or 68K assembly-language programming	C, C++, and Assembly Reference; MSL C Reference; MSL C++ Reference
Pascal or Object Pascal programming	Pascal Language Manual; Pascal Library Reference
Fixing compiler and linker errors	Errors Reference
Fixing memory bugs	ZoneRanger Manual
Speeding up your programs	Profiler Manual
PowerPlant	The PowerPlant Book; PowerPlant Advanced Topics; PowerPlant reference documents
Creating a PowerPlant visual interface	Constructor Manual
Creating a Java visual interface	Constructor for Java Manual
Learning how to program for the Mac OS	Discover Programming for Macintosh
Learning how to program in Java	Discover Programming for Java
Contacting Metrowerks about registration, sales, and licensing	Quick Start Guide
Contacting Metrowerks about problems and suggestions using CodeWarrior software	email Report Forms in the Release Notes folder
Sample programs and examples	CodeWarrior Examples folder; The PowerPlant Book; PowerPlant Advanced Topics; Tutorials (Apple Guide)
Problems other CodeWarrior users have solved	Internet newsgroup [docs] folder

Revision code 970408-CBC-MF